Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andrej Furlan

Deduplikativni datotečni sistemi

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana 2014

Diplomsko delo je stavljeno s sistemom LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Datotečni sistemi so kljub svoji starosti še vedno živahno raziskovalno področje. Pri tem je deduplikacija podatkov, pri kateri optimiziramo hrambo enakih podatkov, eden izmed pomembnejših mehanizmov, ki se pojavlja v sodobnih datotečnih sistemih.

V diplomski nalogi se osredotočite na deduplikativne datotečne sisteme. Preglejte nekaj obstoječih aktivnih implementacij. Nato implementirajte deduplikativni datotečni sistem, pri čemer poskušajte uporabiti nove in drugačne pristope. Na koncu izvedite še primerjavo vašega datotečnega sistema z enim izmed pomembnejših že obstoječih.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Furlan, z vpisno številko **63020041**, sem avtor diplomskega dela z naslovom:

Deduplikativni datotečni sistemi

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Jurija Miheliča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 19. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Juriju Miheliču za vso strokovno pomoč, nasvete, komentarje, popravke, ideje, predloge in pripombe.

Diplomsko delo posvečam Živi F. S.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ACID	Atomicity, Consistency, Isolation, Durability	atomarnost, skladnost, izolacija, trajanje
API	Application Programming Interface	aplikacijski programski vmesnik
BD	Berkeley Database	Podatkovna baza Berkeley
COW	copy-on-write	kopiraj ob pisanju
GNU	GNU's Not Unix	GNU "ni" Unix
GPL	GNU General Public License	Splošno dovoljenje GNU
LGPL	GNU Lesser General Public License	Manj splošno dovoljenje GNU
OS	Operating System	operacijski sistem

Povzetek

V diplomskem delu je predstavljeno področje deduplikacije podatkov. Opisani so primere uporabe in koristi odstranjevanja ali preprečevanja podvojenih podatkov. Razloženi so osnovni koncepti, povezani okrog področja deduplikacije in predstavljeni glavni inženirski problemi, na katere naletimo pri snovanju ali uporabi deduplikativnih sistemov. Glavni poudarek je na deduplikaciji podatkov na podatkovnih medijih; to je deduplikaciji na nivoju datotečnih sistemov. Predstavljeni in primerjani so nekateri glavni obstoječi reprezentativni in med seboj dokaj različni deduplikativni datotečni sistemi. Posebej smo osredotočeni na datotečne sisteme tipa Unix in koncept razvoja deduplikativnih datotečnih sistemov v uporabniškem programskem prostoru. Zasnovan in implementiran je lasten deduplikativni datotečni sistem v uporabniškem programskem prostoru. Z namenom reševanja specifičnega problema deduplikacije je datotečni sistem zasnovan z nekaj posebnimi deduplikativnimi prijemi, ki jih sicer pri obstoječih deduplikativnih sistemih ne najdemo. Deduplikacija je izvedena na nivoju datotek, kar je precejšna redkost med trenutnimi aktualnimi obstoječimi datotečnimi sistemi. Kandidati za deduplikacijo se izbirajo na podlagi imen datotek in informacij o trenutno odprtih datotekah. Implementirani datotečni sistem je konceptualno in eksperimentalno primerjan z odprtokodnim deduplikativnim datotečnim sistemom lessfs in ovrednotene so arhitekturne odločitve obeh sistemov.

Ključne besede: deduplikacija, datotečni sistem, kopiraj ob pisanju.

Abstract

In this thesis the area of data deduplication is presented. Different use cases and benefits of elimination or prevention of duplicate data are described. Basic concepts of data deduplication are explained along with the main engineering challenges faced when designing or using deduplication systems. The main emphasis is on storage-media data deduplication, i.e., data deduplication at filesystem level. A few existing and very different deduplication filesystems are presented and compared. Special focus is given to Unix filesystems and development of userspace deduplication filesystems — in this thesis such a filesystem is designed and implemented. For the purpose of solving a very specific deduplication use case it is designed using a few unique deduplication techniques that are usually not used at existing deduplication filesystems. Deduplication process is implemented at file level, which is rare among the current existing deduplication filesystems. Selection of deduplication candidates is based on file names and a list of currently open files. The implemented filesystem is conceptually and experimentally compared to the opensource deduplication filesystem lessfs. Architectural decisions of both filesystems are discussed.

Keywords: deduplication, filesystem, copy-on-write.

Poglavje 1

Uvod

Deduplikativni datotečni sistemi imajo vgrajene mehanizme za optimizacijo shranjevanja podvojenih podatkov. Klasičen in splošno uporabljan način optimizacije shranjevanja podvojenih podatkov je stiskanje podatkov. Vendar stiskanje podatkov na podatke gleda zelo ozko, ponavadi v okviru ene podatkovne enote – datoteke. Ozko gledano je stiskanje podatkov lahko zelo učinkovito, hitro pa se najdejo primeri, ko stiskanje odpove in spregleda ogromne količine podvojenih podatkov. Deduplikacija je tehnika odstranjevanja podvojenih podatkov, ki na podatke gleda bolj celostno, deluje na višjem nivoju preko meja datotek, na bolj inteligenten način.

1.1 Motivacija

Osnovna motivacija za implementacijo deduplikativnega datotečnega sistema je bila narediti datotečni sistem za inženirje in razvijalce, ki delajo z veliko kopijami enih in istih imeniških dreves izvorne kode (angl. *source tree*).

Konkreten primer je jedro Linux. Trenutna različica jedra Linux (3.16.2 – september 2014) zasede v stisnjeni (`.xz`) obliki 77 MB, v nestisnjeni obliki pa celotna imeniška struktura zasede 528 MB. Na datotečnem sistemu sicer zasede še malo več; veliko k temu doprinese tudi velika količina majhnih datotek. Na datotečnem sistemu `ext4` z velikostjo bloka 4 KB celotno imeniško

drevo jedra Linux zasede 634 MB. Oceno zasedenosti lahko dobimo na sistemih Unix s programom `du`.

Pogosto imajo inženirji v nekem trenutku na datotečnem sistemu več kopij imeniških dreves izvorne kode, na katerih delajo hkrati, npr. za vsak produkt ali vsako različico strojne opreme eno kopijo. Več kopij zaseda več prostora in sicer n kopij zasede n-krat večjo količino prostora. Zelo pogosto pri velikih imeniških drevesih izvorne kode, kot je npr. jedro Linux, inženirji delajo le spremembe na zelo majhni podmnožici vseh izvornih datotek, npr. na eni, dveh, morda desetih datotekah. Vse ostale datoteke v nekem imeniškem drevesu izvorne kode so zelo verjetno enake pripadajočim datotekam v drugih imeniških drevesih izvorne kode. Pojavlja se vprašanje, ali bi lahko morda nekako prihranili ves ta prostor, ki ga zasedajo podvojene povsem enake datoteke.

Danes se velikost pomnilniških medijev, konkretno tu mislimo predvsem na trde diske, izjemno hitro povečuje. Že za relativno majhno ceno je možno dobiti trdi disk z ogromno prostora. Kljub temu se vseeno dostikrat zgodi, da prostora vseeno lahko primanjkuje. Inženirji lahko npr. kodo prevajajo na nekem strežniku tipa Unix v omrežju, kjer si vsak znotraj svojega domačega imenika skopira več kopij imeniškega drevesa jedra Linux. Če je inženirjev veliko, se tudi poraba prostora seveda močno poveča. Strežnik ima lahko omejen prostor že zato, ker je lahko namenjen veliko ljudem in veliko različnim možnostim uporabe, ne npr. samo prevajanju kode. Prostor ima lahko omejen tudi npr. zaradi starejše strojne opreme. Čeprav v splošnem živimo v času, ko je cena pomnilniškega prostora relativno majhna, se torej še vedno lahko pojavljajo primeri, ko se zgodi, da prostora lahko primanjkuje. Povsem na mestu je torej vprašanje, ali lahko kaj prostora na kakšen način prihranimo. Ena izmed možnosti prihranka prostora je torej deduplikativni datotečni sistem.

1.2 Pregled vsebine

V poglavju 2 najprej razložimo osnovne koncepte, ki so povezani z deduplikacijo, konkretneje z deduplikacijo na nivoju datotečnih sistemov oz. shranjevanja podatkov.

V razdelku 2.1 razložimo pojem datotečnega sistema. Pojasnimo, zakaj so datotečni sistemi potrebni, kaj je njihova funkcija in kako iz abstraktnega stališča izgledajo vsi datotečni sistemi. Opišemo, v čem so si datotečni sistemi podobni in v čem se razlikujejo. V razdelku 2.2 se natančneje posvetimo značilnostim in posebnostim datotečnih sistemov tipa Unix. V razdelku 2.3 pojasnimo razlike med klasičnimi datotečnimi sistemi, implementiranimi v prostoru jedra in med datotečnimi sistemi v uporabniškem programskem prostoru. V razdelku 2.4 se posebej posvetimo principom in arhitekturi datotečnih sistemov v uporabniškem programskem prostoru in ogrodju FUSE za razvoj takih sistemov.

V razdelkih 2.6 in 2.7 uvedemo pojem deduplikacije in utemeljimo razloge za njeno uporabo. Razložimo razliko med deduplikacijo in ostalimi klasičnimi načini optimizacije hrambe podatkov. Opišemo različne vrste deduplikacije in inženirske pristope k identifikaciji podvojenih podatkov. Posebej se posvetimo vprašanju računanja prstnih odtisov podatkov z zgoščevalnimi funkcijami. V razdelku 2.8 posebej razložimo mehanizem kopiraj ob pisanju.

V poglavju 3 pregledamo nekaj obstoječih in reprezentativnih deduplikativnih sistemov, tako tistih, implementiranih v prostoru jedra, kot tistih v uporabniškem programskem prostoru.

V poglavju 4 se posvetimo implementaciji lastnega deduplikativnega datotečnega sistema v uporabniškem programskem prostoru. V razdelku 4.1 podamo njegove osnovne lastnosti in utemeljimo razloge, zakaj smo se za take lastnosti odločili. V razdelku 4.2 najprej razložimo osnovne principe razvoja datotečnih sistemov v uporabniškem programskem prostoru z uporabo ogrodja FUSE. Nato razložimo inženirske prijeme in arhitekturne odločitve pri implementaciji zaledja shranjevanja podatkov našega datotečnega sistema. V razdelku 4.3 razložimo principe delovanja logike našega datotečnega

sistema, kot npr., kaj točno se zgodi, ko datoteko ustvarimo, iz nje beremo ali vanjo pišemo. Natančneje se posvetimo opisu implementacije pisanja v datoteko in opisom izvedbe mehanizmov deduplikacije, identifikacije podatkovnih kandidatov za deduplikacijo in izvedbe mehanizma kopiraj ob pisanju.

V poglavju 5 primerjamo implementirani deduplikativni datotečni sistem z obstoječim deduplikativnim datotečnim sistemom lessfs. V razdelku 5.1 ju najprej primerjamo teoretično oz. konceptualno. Primerjamo ju s stališča nivoja, na katerem deduplikacijo izvajata, načina identifikacije enakih blokov in izbire zaledja shranjevanja podatkov. Pri tem utemeljimo arhitekturne odločitve našega datotečnega sistema in teoretično ovrednotimo arhitekturne odločitve lessfs. V razdelku 5.2 nato datotečna sistema primerjamo še eksperimentalno. Izvedemo štiri različne eksperimente, ki vključujejo snemanje več različnih kopij velikih imeniških dreves izvorne kode, snemanja velikih (video) datotek in test brisanja datotek. Vsakič podrobneje opišemo in primerjalno ovrednotimo rezultate ter jih teoretično razložimo.

V poglavju 6 ovrednotimo rezultate eksperimentov oz. testiranj našega datotečnega sistema in predlagamo nekaj možnih izboljšav za nadaljne delo. Nazadnje se preroško zazremo v prihodnost razvoja deduplikativnih sistemov.

Poglavje 2

Osnovni koncepti

2.1 Datotečni sistemi

Datotečni sistem predstavlja način, kako dostopamo do podatkov in v mnogih primerih tudi način, kako so podatki shranjeni oziroma strukturirani na podatkovnem mediju. Na podatkovnih nosilcih (trdi disk, disketa, optični disk, pomnilnik flash) so podatki zapisani seveda strukturirano, v nekem redu in po nekih pravilih. Ta pravila implementira datotečni sistem in ker so podatki fizično ponavadi shranjeni na relativno kompleksen način, datotečni sistem uporabniku nudi nek relativno preprost vmesnik dostopa do podatkov.

Vmesnik dostopa do podatkov pri datotečnem sistemu izhaja iz analogije v realnem svetu – kartotečnih omaric s predalčki, v katerih so kartoteke oziroma mape. Datotečni sistem podatke razdeli na sklope – datoteke. *Datoteka* je linearen skupek podatkov, ki (navadno) vsebinsko sodijo skupaj. Vsaka datoteka ima ime, s katerim se nanjo sklicujemo, in pa pripadajoče meta podatke: npr. velikost datoteke, čas zadnje spremembe, čas zadnjega dostopa, informacije o lastništvu datoteke, tipu datoteke, ipd. Datoteke so (pri večini datotečnih sistemov, ne pa čisto vseh) shranjene hirearhično v *imenikih*[1]. Imeniki lahko vsebujejo datoteke in druge imenike. Na vrhu je korenski imenik.

[1]Nekateri operacijski sistemi za imenik uporabljo izraz mapa (angl. *folder*).

Vmesnik dostopa do podatkov (datoteke, imeniki, itd) je pri vseh datotečnih sistemih bolj ali manj enak oz. podoben z nekaterimi manjšimi razlikami zaradi posebnih lastnosti, ki jih datotečni sistem implementira ali operacijskega sistema, kateremu je namenjen. Razlike so lahko npr. v tem, kakšni znaki so dovoljeni v imenu datoteke, koliko je največja dolžina datoteke, kakšne lastnosti datoteke sistem izpostavlja, največje število datotek na datotečnem sistemu, ipd. Konceptualno pa vsi datotečni sistemi izgledajo enako – podatki so enkapsulirani v datotekah, te pa so večinoma hirearhično razporejeno po imenikih.

Nasprotno pa se datotečni sistemi lahko zelo razlikujejo v sami izvedbi shranjevanja oziroma v splošnem v načinu dostopanja do podatkov. Razlike v strukturi oziroma načinu shranjevanja podatkov med datotečnimi sistemi izhajajo iz zgodovinskih razlogov (uporabljene podatkovne strukture in principi sledijo novim spoznanjem v računalniški znanosti), posebnih lastnosti, ki jih datotečni sistem ima ali izpostavlja (hitrost, varnost, dnevniški zapisi, preprečevanje podvajanja podatkov itd.) ali pa preprosto razlike med posameznimi proizvajalci ali operacijskimi sistemi, za katere je nek datotečni sistem namenjen.

Pogosto so razlike v izvedbi shranjevanja podatkov posledica specifičnih lastnosti medija, za katerega je nek datotečni sistem primarno namenjen: npr. datotečni sistem jffs2 je namenjen bliskovnim pomnilnikom (angl. *flash memory*) in diskom SSD[2] (ki uporabljajo tehnologijo bliskovnega pomnilnika). Pri tehnologiji bliskovnega pomnilnika je medij razdeljen na segmente, ki so individualno izbrisljivi, število brisalnih ciklov pa je navzgor omejeno: tipično 3000 do 5000 ciklov preden dostop do segmenta postane nezanesljiv. jffs2 in podobni datotečni sistemi zato implementirajo t.i. tehniko izravnave obrabe (angl. *wear leveling*) [13], ki skrbi, da se vsi segmenti na celotnem mediju približno enakomerno obrabljajo, kar podaljšuje življensko dobo medija.

V osnovi so datotečni sistemi taki, da neposredno dostopajo do fizičnih podatkov na podatkovnem mediju. Lahko pa datotečni sistem do podat-

[2]SSD – Negibljivi pogon (angl. *Solid State Drive*).

kov dostopa tudi posredno, npr. preko omrežja, kot je to pri omrežnih datotečnih sistemih – najbolj znan predstavnik je morda NFS (angl. *Network File System*). Omrežni datotečni sistemi omogočajo dostop do datotek preko omrežja na nekem strežniku, vmesnik za dostop pa je povsem enak kot pri običajnih datotečnih sistemih, zato se nam zdi, da z datotekami delamo lokalno, v resnici pa datoteke dejansko živijo na nekem strežniku.

Datotečni sistem je lahko tudi navidezen. Pri tem datoteke in imeniki, ki jih prikazuje uporabniku, pravzaprav niso prave datoteke oz. imeniki ampak samo način pogleda na neke podatke. Primer je datotečni sistem `procfs` v operacijskih sistemih Unix, kjer preko hirearhične ureditve imenikov in datotek lahko beremo in nastavljamo trenutne lastnosti procesov oz. jedra operacijskega sistema, tako da beremo ali pišemo v neko datoteko znotraj `procfs`. Če poznamo številko procesa (PID – process identifier), lahko npr. izvemo, katere datoteke ima proces s številko PID `<pid>` odprte tako, da preberemo vsebino imenika `/proc/<pid>/fd` – npr. z ukazom `ls -l /proc/<pid>/fd`. Tak datotečni sistem je torej le abstraktni uporabniški vmesnik v obliki hirearhične strukture imenikov in datotek običajnega datotečnega sistema.

Način, kako je datotečni sistem dosegljiv uporabniku (oz. programu), je odvisen od operacijskega sistema. V okoljih Microsoft Windows operacijski sistem vsakemu datotečnemu sistemu dodeli črko pogona – tipično se za primarni razdelek primarnega trdega diska uporablja oznaka `C:`, za (danes sicer ne več aktualne) diskete pa oznaki `A:` ali `B:`. Tako npr. `C:\Program Files` predstavlja imenik `Program Files` na pogonu `C:`. Drugače je pri operacijskih sistemih tipa Unix, tu operacijski sistem predstavi vse datotečne sisteme v eni veliki imeniški hirearhiji – korenskem datotečnem sistemu. Na vrhu je korenski imenik `/`, pod njim pa hirearhično ostali imeniki, ti pa lahko predstavljajo dejanske imenike na različnih podatkovnih medijih (sistemski trdi disk oz. njegovi razdelki, optične enote, enote USB, ipd) – rečemo, da so podatkovni mediji (oz. njihovi razdelki – del podatkovnega medija, ki ga operacijski sistem obravnava kot ločeno enoto) priklopljene (angl. *mount*) na nek imenik v datotečnem sistemu.

Operacijski sistem za uporabo datotečnih sistemov nudi uporabniški programski vmesnik (API – angl. *Application Programming Interface*), ki je abstrakten in enak za vse datotečne sisteme, ki jih lahko s tem operacijskim sistemom uporabljamo. API vključuje funkcije za odpiranje in zapiranje datotek, branje in pisanje datotek, branje imenikov (njihove vsebine – seznama datotek, ki jih vsebujejo), premik trenutne pozicije branja/pisanja v datoteki, ustvarjanje datotek, ogled meta podatkov o datotekah ipd. Pri sistemih Unix je API realiziran v obliki sistemskih klicev (`open, close, read, write, readdir, lseek, stat` itd).

Datotečni sistemi so lahko dnevniški (angl. *journaling*) ali običajni nednevniški. Pri dnevniški datotečnih sistemih se vsaka sprememba izvede tako, da se najprej v dnevnik (angl. *log, journal*) zapiše opis te spremembe, šele nato se sprememba dejansko izvede. V primeru izpada električne energije ali sesutja sistema je morebitna obnovitev datototečnega sistema precej enostavnejša in hitrejša.

2.2 Datotečni sistemi Unix

V tej diplomski nalogi se bomo posvetili predvsem datotečnim sistemom namenjenim operacijskim sistemom tipa Unix. Deduplikativni datotečni sistem, ki ga bomo implementirali, bo narejen za operacijski sistem GNU Linux.

Pomembna značilnost operacijskih sistemov tipa Unix je, da je datotečni sistem zelo pomemben, pravzaprav celo integralni del operacijskega sistema. Operacijski sistem (OS) namreč vso strojno opremo abstrahira s posebnimi vrstami datotek – datoteke naprav (angl. *device files*). To niso klasične datoteke, saj nimajo pripadajočih podatkovnih blokov na nekem podatkovnem mediju, ampak so samo vnos v datotečnem sistemu, poimenovanje za napravo, ki ga uporablja programska oprema, ko se na neko strojno opremo sklicuje oz. jo uporablja. Tudi mnogi drugi osnovni mehanizmi operacijskega sistema, kot npr. medprocesna komunikacija, so abstrahirani z datotekami. Na primer, vtičnice domene Unix (angl. *Unix domain sockets*), eden izmed

mehanizmov medprocesne komunikacije, uporabljajo datoteke oz. poti v datotečnem sistemu kot komunikacijski naslov bodisi odjemalca ali strežnika. Tudi tu je datoteka le vnos v datotečnem sistemu in ni povezan s podatki na kakršnemkoli podatkovnem mediju.

Prednost takega pristopa je, da programi uporabljajo enak programski vmesnik tako za delo z običajnimi datotekami kot za delo s posebnimi datotekami naprav, vtičnic, cevi, ipd. Dva programa, ki komunicirata preko vtičnice domene Unix, odpreta ustrezno posebno datoteko in na njej izvajata pisalne (write) in bralne (read) operacije, kot da bi šlo za pravo datoteko. Seveda se, ker gre le za abstrakcijo, podatki nikakor ne prenašajo preko kakršnegakoli podatkovnega medija (recimo trdega diska), ampak se prenašajo direktno med procesoma (oziroma vmes zavijejo v jedro OS).

Pri OS Unix je vsaka datoteka predstavljena s številko *inode*, ki je v bistvu referenca na metapodatke o datoteki (oziroma vnos inode). Metapodatki o datoteki vsebujejo npr. podatke o tipu datoteke (običajna datoteka, imenik, datoteka naprave, vtičnica, cev, itd.), lastništvu datoteke (uporabnik, skupina), vrste dovoljenega dostopa (branje, pisanje, izvajanje – v primeru binarnih izvršljivih datotek), velikost datoteke, časovne informacije (čas zadnjega dostopa, zadnje spremembe vsebine, zadnje spremembe metapodatkov) ipd. Metapodatki v vnosu inode vsebujejo tudi reference oziroma informacije o lokacijah blokov fizičnih podatkov na podatkovnem mediju. Vnos inode pa ne vsebuje imena datoteke. Preslikave imena datotek v številke inode so zapisane ločeno.

Imenik je le posebne vrste datoteka (torej imajo tudi imeniki svoj pripadajoči vnos inode). V grobem rečeno, imeniki vsebujejo seznam preslikav imen vsebovanih datotek v njihove številke inode.

Datoteka je torej predstavljena z vnosom inode, vsak vnos inode (in s tem posledično datoteka) pa ima lahko več kot eno ime, čemur rečemo trda povezava (angl. *hard link*). Če imamo npr. datoteko `/home/foo/bar.txt` lahko naredimo dodatno ime BAR.txt za to datoteko tako, da naredimo trdo povezavo na to datoteko z `ln /home/foo/bar.txt /home/foo/bla/BAR.txt`.

Obe imeni sta enakovredni, čeprav je bilo eno ustvarjeno kasneje, in kažeta na isti vnos inode, torej predstavljata isto datoteko. Da zbrišemo datoteko, moramo zbrisati "obe datoteki" – dejansko zbrišemo obe imeni datotek oziroma natančeneje rečeno, zbrišemo obe trdi povezavi na to datoteko. Zaradi tega se proces brisanja pri operacijskih sistemih tipa Unix pravzaprav imenuje prekinjanje povezave (angl. *unlink*).

Datoteko lahko zbrišemo tudi, ko jo ima še vedno eden ali več procesov odprtih. Pri tem se zbriše le povezava imena datoteke v vnos inode in ker proces uporablja ime datoteke samo pri odpiranju datoteke, nadalje pa se sklicuje na datoteko (posredno) preko njenega vnosa inode, ima lahko proces "zbrisano" datoteko odprto še poljubno dolgo. Datotečni sistem fizične podatke datoteke na podatkovnem mediju (in seveda njene meta podatke) dejansko zbriše šele, ko datoteko zapre zadnji proces in ko število trdih linkov datoteke pade na 0.

Zaradi same narave trdih povezav te lahko kažejo le na datoteke znotraj istega datotečnega sistema oziroma razdelka. Večina operacijskih sistemov omogoča trde povezave le do datotek, ne pa tudi do imenikov, da preprečimo cikle.

Poleg trdih povezav so možne tudi simbolične povezave (angl. *symbolic link*). Simbolična povezava je datoteka, ki kaže na neko drugo datoteko ali imenik. Simbolična povezava je v resnici običajna datoteka (s svojo številko inode), njena vsebina pa je niz – pot do druge datoteke, pri čemer ni nujno, da ciljna datoteka obstaja. Ciljna datoteka je lahko tudi na drugem razdelku oziroma drugem datotečnem sistemu.

2.3 Izvedba v uporabniškem prostoru

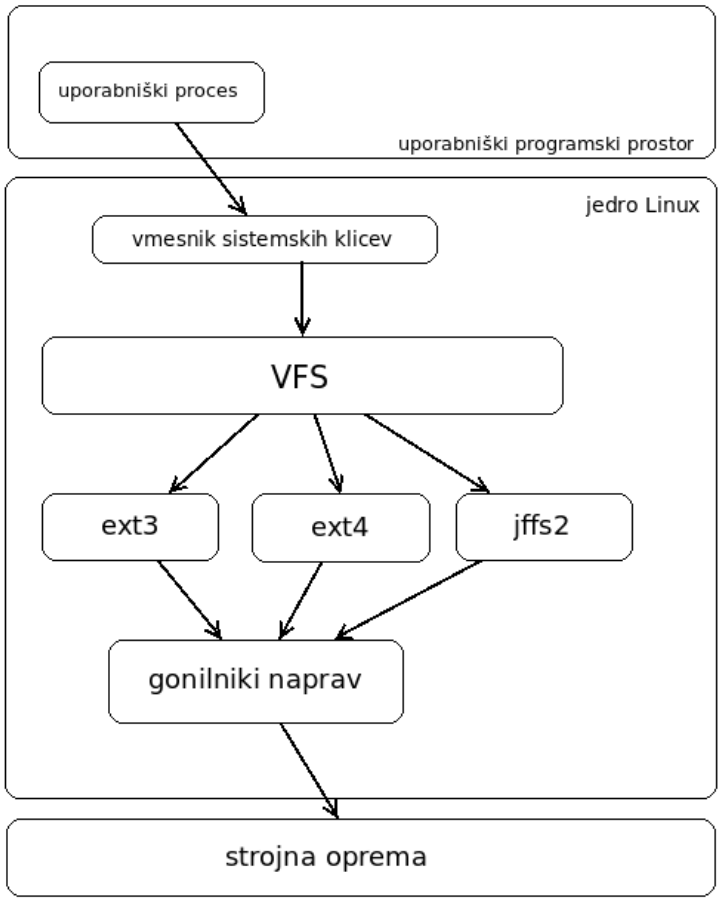
Kot že omenjeno se implementacija datotečnih sistemov navadno nahaja znotraj jedra operacijskega sistema. Vsekakor to velja za velike in splošne datotečne sisteme. Manjši ali pa datotečni sistemi, namenjeni bolj specifičnim potrebam, so pogosto implementirani v uporabniškem programskem pro-

storu, torej zunaj jedra operacijskega sistema.

Večina modernih operacijskih sistemov namreč izvaja kodo jedra in gonilnikov naprav v priviligiranem načinu procesorja, kjer so omogočeni vsi ukazi procesorja, vso ostalo kodo pa v nepriviligiranem načinu procesorja, kjer so nekateri sistemski ukazi procesorja onemogočeni (npr. nastavljanje prekinitev, nastavitve virtualnega pomnilnika, ipd.). Tako se zagotovi varnost, saj v primeru sesutja slabo napisanega uporabniškega programa ta ne more, ali pa vsaj veliko težje, ogrozi oziroma sesuje celoten sistem. Le jedro ima neposreden dostop do strojne opreme. Procesi, ki tečejo v uporabniškem prostoru, uporabljajo storitve jedra tako, da izvedejo sistemski klic. Sistemski klici so API jedra za njegove storitve.

Implementacija datotečnega sistema načeloma sodi v jedro operacijskega sistema. Jedro navadno tudi poskrbi za programski vmesnik, ki ga koda datotečnega sistema mora izpolnjevati. Takemu programskemu vmesniku rečemo virtualni datotečni sistem – angl. *Virtual File System* (VFS). Namen takega vmesnika je abstrakcija različnih vrst datotečnih sistemov, vmesni nivo nad konkretnimi datotečnimi sistemi. Koda, ki kakorkoli dela z datotečnimi sistemi, tako uporablja različne datotečne sisteme na enovit način. Za shematični prikaz glej sliko 2.1. Uporabniški proces interaktira z VFS posredno preko vmesnika sistemskih klicev, ki je most med uporabniškim programskim prostorom in prostorom jedra. VFS je vmesnik oz. abstrakcija za konkretne datotečne sisteme, npr. ext3, ext4 ali jffs2. Ti do strojne opreme (npr. trdega diska) dostopajo posredno preko gonilnikov naprav.

V jedro sodijo splošnonamenski datotečni sistemi. Datotečni sistemi, ki rešujejo specifične probleme, v jedro ne sodijo že za to, ker niso dovolj splošni. Dodatno je distribucija in namestitev programske opreme, ki vsebuje kodo, ki naj bi tekla znotraj jedra, veliko zahtevnejša. Ker je operacijski sistem zasnovan tako, da predvideva implementacijo datotečnega sistema znotraj jedra (preko VFS), moramo, če želimo implementirati datotečni sistem zunaj jedra, uporabiti določene trike, da obvozimo to osnovno predpostavko. Te trike nam elegantno omogočajo nekatera programska ogrodja za programi-



Slika 2.1: Shema relacij med uporabniškim programskim prostorom, jedrom Linux, VFS, konkretnimi datotečnimi sistemi ter strojno opremo.

ranje datotečnih sistemov v uporabniškem programskem prostoru. Najbolj znano tako ogrodje je FUSE, ki se mu bomo natančneje posvetili v naslednjem poglavju in tudi naš deduplikativni datotečni sistem bo narejen s pomočjo tega ogrodja. Obstajajo sicer tudi druga ogrodja, vendar so precej manj znana in razširjena, kot npr. LUFS in UserFS [16]. Za sisteme Windows obstaja npr. ogrodje Dokan [20].

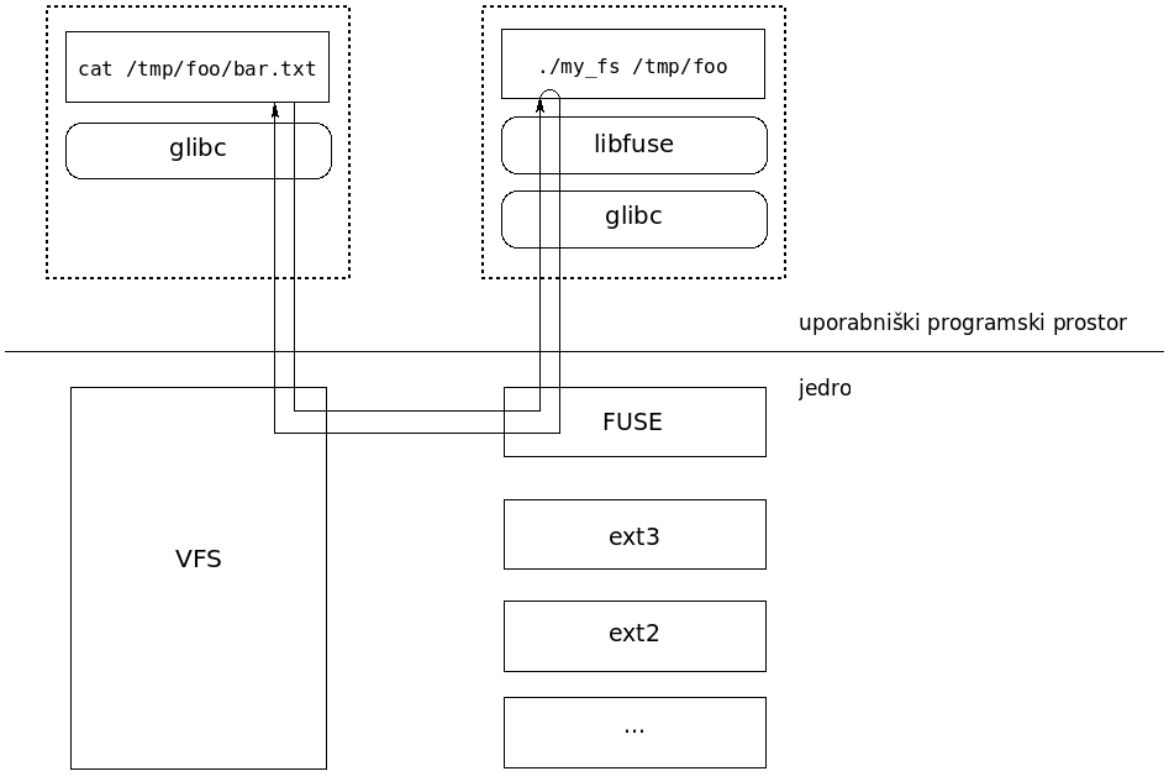
2.4 Ogrodje FUSE

FUSE je programsko ogrodje za razvoj datotečnih sistemov v uporabniškem programskem prostoru za operacijske sisteme tipa Unix. Originalno zasnovan za OS Linux, kasneje pa prenesen oziroma implementiran še za nekatere druge operacijske sisteme tipa Unix: npr. FreeBSD, NetBSD, Mac OS X, Open Solaris [17].

S pomočjo FUSE lahko razvijemo svoj datotečni sistem, ne da bi nam bilo treba pisati kodo, ki bi tekla znotraj jedra OS. To pomeni, da je distribucija in namestitev datotečnega sistema izvedenega s FUSE veliko enostavnejša. Enaka ali morda še bolj pomembna posledica je tudi, da lahko datotečne sisteme tipa FUSE priklopi tudi navadni neprivilegirani uporabnik. Zaradi varnostnih razlogov namreč lahko običajne datotečne sisteme priklaplja le priviligirani uporabnik (root).

Razvoj novega datotečnega sistema poteka tako, da napišemo program, ki se povezuje z deljeno knjižnico FUSE in ki implementira določene funkcije, ki jih FUSE zahteva in se kličejo po metodi povratnih klicev (angl. *callback*). Večina funkcij za povratne klice se konceptualno preslika v običajne sistemske klice za delo z datotečnimi sistemi – `open`, `read`, `write`, itd. Le da tu delamo na višjem nivoju, ker za večino nizkonivojskih podrobnosti poskrbi že samo ogrodje FUSE. Datotečni sistem je torej implementiran kot proces, ki teče v uporabniškem programskem prostoru.

Na sliki 2.2 je prikazana arhitektura FUSE in delovanje datotečnega sistema implementiranega s FUSE. `./my_fs` je program, ki implementira



Slika 2.2: Prikaz arhitekture in delovanja FUSE (Shema povzeta po [18]).

FUSE in z ukazom `./my_fs /tmp/foo` smo priklopili naš datotečni sistem na imenik `/tmp/foo`. Hkrati smo pognali proces `my_fs`, ki sedaj teče v ozadju. libfuse je knjižnica FUSE, glibc pa standardna knjižnica C. Ukaz `cat /tmp/foo/bar.txt` izpiše vsebino datoteke `bar.txt` in ker se ta datoteka nahaja na poti, ki je priklopljena z datotečnim sistemom tipa FUSE, se sistemski klici (npr. `read`), ki se tičejo tega datotečnega sistema, preusmerijo preko VFS v jedrni del FUSE. Ta je bodisi vgrajen v jedro ali pa izveden kot jedrni modul. Ta zahtevo posreduje našemu programu, ki implementira datotečni sistem – izvede se povratni klic, ki ustreza izvedenemu sistemskemu klicu in rezultati se preko libfuse posredujejo naprej v jedrni modul FUSE, potem preko VFS nazaj v program `cat`. Vidimo, da program `cat` z našim datotečnim sistemom dela na povsem enak način kot z ostalimi datotečnimi sistemi – preko VFS. V ozadju pa FUSE prestreza zahtevke in jih posreduje našemu programu, ki implementira datotečni sistem. Komunikacija med jedrom in uporabniškim programskim prostorom poteka preko posebne datoteke naprave `/dev/fuse`.

FUSE vsebuje tudi orodje `fusermount` za prikloplanje in odklapljanje datotečnih sistemov FUSE. Knjižnica libfuse je licencirana z LGPL, ostala koda FUSE pa z GPL.

Ogrodje FUSE je pisano v programskem jeziku C, ki je tudi domorodni programski jezik za uporabo tega ogrodja. Omogoča pa tudi jezikovne povezave (angl. *language bindings*) za kopico drugih jezikov, kot npr. C++, Python in Java.

2.5 Primeri posebnih datotečnih sistemov

Nekateri datotečni sistemi so zelo specifični, rešujejo nek konkreten ozko usmerjen problem. `sshfs` omogoča dostop do datotek na nekem oddaljenem strežniku, pri čemer se nam zdi, kot da so datoteke dostopne na našem lokalnem računalniku. V resnici so to iste datoteke, do katerih bi sicer imeli dostop preko terminalske seje.

Datotečni sistem `truecrypt` omogoča transparento šifriranje podatkov v realnem času in njihovo predstavitev v obliki datotek oz. imenikov pri čemer se (šifrirani) podatki lahko dejansko zapisujejo na namenski razdelek ali pa npr. kar v posebno datoteko na obstoječem datotečnem sistemu.

`soundcloudFS` je datotečni sistem, ki omogoča, da glasbene tokove (angl. *sound stream*), ki jih sicer pri spletni storitvi SoundCloud lahko poslušamo le preko spletne strani, v spletnem brskalniku poslušamo lokalno s poljubnim lokalnim predvajalnikom, pri čemer so glasbeni tokovi predstavljeni kot glasbene datoteke na datotečnem sistemu.

`archivemount` omogoča, da vsebino arhivskih datotek (npr. .tar. tar.gz, .zip) vidimo kot datoteke in imenike na datotečnem sistemu, ne da bi nam bilo potrebno arhivske datoteke odpakirati.

2.6 Deduplikacija

Deduplikacija je tehnika odstranjevanja podvojenih podatkov. S tem dosežemo manjšo porabo prostora na podatkovnih medijih ali pa manjšo porabo pasovne širine, če gre za prenos podatkov, npr. po omrežju.

Prvi primeri uporabe deduplikacije so bili realizirani pri sistemih za varnostno kopiranje oziroma strežnikih za shranjevanje varnostnih kopij, saj tam navadno nastaja največ podvojenih podatkov. Vsako inkrementalno varnostno kopiranje lahko ustvari velike količine podvojenih podatkov, ki zato po nepotrebnem zasedajo veliko prostora. Strežniki za elektronsko pošto pogosto shranjujejo veliko količino podvojenih podatkov – ko npr. nekdo pošlje e-pošto z relativno veliko priponko (npr. 5 MB) vsem (ali pa veliko) sodelavcem znotraj nekega podjetja, e-poštni strežnik shranjuje enako kopijo datoteke v poštnem predalu vsakega prejemnika.

V zadnjem času je zelo popularna virtualizacija. Pri tem na enem gostiteljskem računalniku gostuje več t.i. navideznih strojev. Ti imajo datotečni sistem virtualiziranega operacijskega sistema shranjen v obliki datoteke – podatkovne slike – na gostiteljskem računalniku. Pogosto en gostitelj poga-

nja več enakih navideznih operacijskih sistemov, ki pa se lahko razlikujejo le npr. v konfiguraciji in morda še v nekaj podatkih na datotečnem sistemu. Vsi pa imajo enake kopije osnovnih sistemskih datotek in programov in te po nepotrebnem lahko zasedajo ogromno odvečnega prostora. V takih primerih lahko deduplikacija prihrani do 40 % prostora, če jo izvajamo nad slikami različnih navideznih strojev in celo do 80 % prostora, če jo izvajamo nad slikami navideznih strojev, ki se razlikujejo le v različici, a so znotraj iste skupine OS (npr. Fedora, Ubuntu) [1].

Nekateri deduplikacijo imenujejo tudi inteligento stiskanje podatkov. Stiskanje podatkov deluje vedno le znotraj posameznih logičnih enot podatkov, npr. datotek. V datotekah se identificirajo zaporedja podatkov, ki se ponavljajo, ti manjši podnizi se potem shranijo drugače in tako na koncu datoteka zasede manj. Postopek stiskanja podatkov vedno gleda naenkrat le znotraj ene logične enote podatkov (npr. datoteke), če imamo v dveh datotekah na nekem mestu enaka kosa podatkov, tega stiskanje ne zazna. Deduplikacija, po drugi strani, deluje na nivoju celotne logične enote medija (npr. trdi disk oz. njegov razdelek) in torej lahko identificira enake datoteke ali pa le posamezne enake kose podatkov znotraj različnih datotek.

Naslednja razlika med stiskanjem podatkov in deduplikacijo je transparentnost. Stiskanje lahko je ali pa tudi ne transparentna za uporabnika. Nekateri datotečni sistemi npr. omogočajo transparentno stiskanje podatkov, večina pa ne. Uporabniki pogosto shranjujejo svoje datoteke tako, da jih zapakirajo v arhivske datoteke, ki jih potem stisnejo s kakšnim izmed standardnih algoritmov za stiskanje podatkov (zip, gzip, bzip2, itd). To vse počnejo ročno, torej takšno stiskanje podatkov ni transparentno.

Bistvo deduplikacije pa je, da je za uporabnika praktično vedno transparentna. Uporabnik npr. naredi novo kopijo datoteke na datotečnem sistemu, deduplikacija vgrajena v sam sistem pa avtomatsko poskrbi, da se dejansko ne naredi prava kopija podatkov, ampak se ustvari samo povezava na že obstoječe podatke. Pri branju druge kopije datotečni sistem pogleda povezavo, ter avtomatsko postreže s podatki, na katere povezava kaže. Ko uporab-

nik začne spreminjati drugo datoteko, datotečni sistem najprej avtomatsko naredi dejansko novo kopijo podatkov, da spremembe na drugi datoteki ne vplivajo na prvo datoteko.

Razlika med stiskanjem podatkov in deduplikacijo je tudi v tem, pri kakšnih dolžini blokov, ki jih metoda gleda naenkrat, je bolj učinkovita. Stiskanje podatkov je bolj učinkovito, če je dolžina blokov večja, deduplikacija pa je bolj učinkovita, če je dolžina blokov manjša.

Mnogi datotečni sistemi že omogočajo določene aspekte deduplikacije, ki pa so vseeno zelo pomankljivi. Primer so trde povezave (angl. *hardlink*), ki jih omogočajo datotečni sistemi tipa Unix. Vendar so trde povezave dobre le, če uspemo zagotoviti, da se bodo duplicirani podatki le brali. Kakršnakoli sprememba podatkov v datoteki, ki ima več kot eno trdo povezavo, se seveda odraža na vseh poteh, ki so povezane s pripadajočim vnosom inode, mehanizma za avtomatsko ustvarjanje nove kopije v takem primeru pa seveda pri trdih povezavah ni.

Tehnike deduplikacije se v osnovi delijo na *vmesne* (angl. *inline*) in *naknadne* (angl. *post-process*). Pri vmesni deduplikaciji sistem izvaja deduplikacijo v istem trenutku kot podatki nastajajo oziroma se spreminjajo. Ko npr. posnamemo na datotečni sistem novo datoteko, sistem v istem trenutku analizira, ali so med prihajajočimi podatki morda enaki kosi podatkov kot že shranjeni. Pri naknadni deduplikaciji se proces analize oziroma deduplikacije izvaja kasneje, ko so podatki že nastali. To se lahko zgodi na zahtevo – npr. sistemski administrator požene posebno orodje, ki analizira podatke na disku in izvede morebitno deduplikacijo, ali pa se dogaja v ozadju – npr. vsake toliko časa se zažene analiza nad podatki in morebiti izvede deduplikacija.

V literaturi vmesni deduplikaciji pogosto rečejo tudi *sinhrona*, naknadni deduplikaciji pa *asinhrona* ali pa *paketna* deduplikacija (angl. *batch mode deduplication*).

Prednost vmesne deduplikacije je, da nikoli ne shranjujemo podvojenih podatkov, niti za krajši čas ne. Slabost pa je lahko manjši ali večji padec zmogljivosti operacij nad podatki. Vsaka operacija nad datotečnim sistemom, ki

podatke dodaja ali jih spreminja, se nujno upočasni, ker sedaj vključuje še analizo podatkov in morebitno deduplikacijo. Prednost naknadne deduplikacije je, da uporabnik ne čuti performančnih razlik pri izvajanju operacij nad podatki datotečnega sistema. Slabost pa, da lahko za krajši ali daljši čas podatke shranjujemo podvojeno, kar je lahko problem, če je zasedenost podatkovnega medija že blizu njegove kapacitete. V tem primeru se lahko zgodi, da nekaterih novih podatkov ne moremo shraniti, čeprav bi po procesu analize in deduplikacije morda kapaciteta podatkovnega medija še ne bi bila zapolnjena.

Večina metod deduplikacije deluje tako, da podatke obravnava po kosih določene velikosti. Morebitni novi podvojeni kosi se ne shranijo še enkrat, ampak se shrani samo referenca na podvojeni podatek, ali pa kakšen drug način identifikacije podatka. Tem kosom podatkov rečemo bloki. Velikost blokov je lahko fiksna (npr. 4 KB ali 128 KB), ali pa spremenljiva. Poseben primer spremenljive dolžine blokov je, če deduplikativni sistem vedno gleda le datoteko kot celoto. V tem primeru deduplikacijo imenujemo tudi angl. *Single Instance Storage*. Izvedba deduplikacije s fiksnimi bloki je seveda enostavnejša kot izvedba s spremenljivimi bloki (razen za poseben primer, ko sistem gleda le celotne datoteke).

Pri manjših blokih je učinkovitost deduplikacije večja (pri manjši velikosti je večja možnost, da najdemo podvojeni blok), hkrati pa se navadno poveča tudi količina meta podatkov. Za vsak blok sistem namreč hrani določene meta podatke (npr. katere datoteke ga uporabljajo in njihov prstni odtis). Zato je treba najti pravo razmerje med učinkovitostjo deduplikacije in količino meta podatkov, ki lahko pri zelo majhnih blokih vseeno zasedejo kar precej prostora.

2.7 Zgoščanje

Blokom podatkov se ponavadi dodeli nek identifikator – prstni odtis podatkov, ki se izračuna s pomočjo kriptografskih zgoščevalnih funkcij (npr. MD5,

SHA-1, SHA-256). Izračunan prstni odtis podatkovnega bloka se primerja s prstnimi odtisi že obstoječih shranjenih blokov in če se najde blok z enakim prstnim odtisom, potem je ta blok kandidat za deduplikacijo. Ker pa prstni odtisi, pridobljeni z zgoščevalnimi funkcijami, niso nujno enolični zaradi možnosti trkov, bi striktno gledano potem morali preverjati še vsebino blokov bajt po bajtu, a večina deduplikativnih sistemov ta korak zaradi hitrostne učinkovitosti izpusti. Za primerjanje vsebine moramo obstoječi blok namreč brati s podatkovnega medija, kar predstavlja dodatne vhodno/izhodno operacije, ki pa so v osnovi počasne. Tako sistem enostavno predpostavi, da sta dva bloka enaka, če imata enak prstni odtis. Ker se seveda lahko zgodi, da imata zaradi narave zgoščevalnih funkcij dva različna bloka enak prstni odtis, seveda to pomeni, da obstaja (sicer zelo majhna) verjetnost izgube ali okvare podatkov. Vendar, če uporabimo dovolj dobro zgoščevalno funkcijo, se da pokazati, da je verjetnost izgube oz. okvare podatkov zaradi zgoščevalne funkcije, ki bi dala enako vrednost za dva različna bloka, manjša kot pa verjetnost nedetektirane strojne napake na podatkovnem mediju [2]. Prstne odtise blokov (njihove zgoščene vrednosti) lahko (v celoti ali delno) predpomnimo v pomnilniku in s s tem pospešimo proces iskranja duplikatov.

Deduplikacija s fiksnimi bloki se slabo obnese, kadar *vstavljamo* ali *brišemo* podatke na sredini oziroma bolj proti začetku obstoječe datoteke in količina vstavljenih podatkov ni večkratnik dolžine bloka. Pri tem namreč pride do zamika mej vseh blokov, ki sledijo oziroma do spremembe obstoječih blokov. Če so bili obstoječi bloki prej duplikati, sedaj najverjetneje niso več. Hkrati moramo tudi ponovno izračunati prstni odtis oz. zgoščeno vrednost novih blokov. Ta problem izhaja iz osnovne narave datotek, ki so definirane kot strogo zaporedje podatkov.

Temu problemu se lahko izognemo z uporabo deduplikacije s spremenljivimi dolžinami blokov in uporabo metode *vsebinsko naravnanega razdeljevanja na bloke* (angl. *content-based chunking*). Pri tej metodi meje med bloki (in s tem dolžino posameznih blokov) določamo glede na vsebino podatkov. Najpogostejša uporabljena metoda je uporaba algorithma *Rabinovo*

zgoščanje (angl. *Rabin fingerprinting*).

Rabinovo zgoščanje uporablja drseče okno konstantne širine w (tipično 48) bajtov. Drseče okno se na vsakem koraku premakne naprej za en bajt. Za vsako drseče okno izračuna *zgoščeno vrednost Rabin*. Ko je ta enaka k nizkim bitom neke vnaprej izbrane konstane, takrat označimo novo mejo med blokoma. Ker lahko taka metoda da zelo različne velikosti blokov, implementacije ponavadi definirajo še minimalno in maksimalno velikost bloka [1, 2, 3]. Rabinovo zgoščanje je natančno opisano v izvornem članku [4]. Za določanje mej med bloki bi sicer lahko uporabili tudi kako drugo zgoščevalno funkcijo; Rabinovo zgoščanje je primerno tudi zato, ker se jo da učinkovito implementirati zaradi lasnosti, da lahko na nekem koraku delno uporabimo že izračunane vrednosti iz prejšnega koraka [12]. Taka lastnost je značilna za t.i. *vrteče zgoščevalne funkcije* (angl. *rolling hash*).

Deduplikacija s spremenljivimi dolžinami blokov lahko torej zelo izboljša učinkovitost deduplikacije, kadar imamo opravka z vstavljanjem ali brisanjem le nekaj bajtov v bližini začetka datotek. Prav tako je ta metoda zelo primerna za podatke, kjer je veliko potencialno koreliranih podatkov, ki jih metoda s fiksnimi bloki ne bi mogla identificirati [3]. Ker bloki niso fiksne dolžine, lahko prihaja do neporavnanih vhodno-izhodnih operacij – začetek bloka ni poravnan na naslov, ki je večkratnih domorodne dolžine bloka, kar lahko negativno vpliva na performance.

V splošnem bi se nam lahko zdelo, da je zaradi svojih lastnosti vsebinsko naravnano razdeljevanje na bloke tista prava izbira, kadar želimo maksimalno učinkovitost deduplikacije. Glede ne njeno odpornost na vstavljanja oziroma brisanja majhnih količin podatkov na začetku datoteke se namreč zdi superiorna metodi fiksne dolžine blokov in to kljub temu, da je precej bolj zahtevna za implementacijo. Pa vendar moramo razmišljati tudi o tem, kako pogosto se take operacije v določenih primerih uporabe oz. pri določenih tipih podatkov izvajajo. Oziroma koliko je ponavljanj v podatkih, ki jih lahko zaznamo le z vsebinsko naravnanim razdeljevanjem na bloke. Pri meritvah na realnih eksperimentih večina raziskovalcev sicer res ugotavlja, da vsebinsko naravnano

razdeljevanje na bloke v večini da najboljše rezultate deduplikacije, hkrati pa tudi ugotavljajo, da se jim metode s fiksnimi bloki pogosto zelo približajo [3] ali v nekaterih primerih celo malenkost presežejo [1].

2.8 Kopiraj ob pisanju

Kopiraj ob pisanju (angl. *copy-on-write* – COW) je optimizacijska tehnika, ki se uporablja na mnogih področjih računalništva. Pogosto moramo narediti kopijo nekih podatkov, hkrati pa obstaja velika verjetnost, da se bo novonastala kopija le brala (ves čas ali pa daljši čas). Kopijo podatkov zato naredimo tako, da ne naredimo prave kopije, ampak bo nova kopija le referenca na obstoječo kopijo in ustrezno označena kot referenca COW. Dokler se vse kopije (oziroma reference) istega podatka le berejo, lahko tako tudi ostane. V trenutku, ko se katerakoli kopija začne spreminjati, najprej naredimo novo dejansko kopijo podatkov, na kateri se potem izvedejo spremembe. Originalna kopija podatkov pa ostane nespremenjena.

Večinoma so operacije COW še dodatno optimizirane tako, da se pri izdelavi dejanske kopije podatkov v resnici kopirajo le podatki, ki so v okviru neke logične enote (blok pri datotečnih sistemih, stran pri glavnem pomnilniku), v okviru katere se sprememba dogaja. Vse ostale logične enote (bloki, strani) ostanejo reference na obstoječe podatke.

Rezultat optimizacije je prihranek tako prostora (reference zasedejo zanemeraljivo količino prostora glede na dejanske podatke) kot časa (ustvarjanje referenc traja zanemarljivo količino časa kot kopiranje celotnih podatkov).

COW se uporablja pri mehanizmu ustvarjanja novih procesov na sistemih Unix. Proces (starš) ustvari nov proces (otrok) s sistemskim klicem `fork`, ki naredi kopijo starševskega procesa, ki je identična staršu v skoraj vsem, razen v nekaterih parametrih. Predvsem pa otrok podeduje oziroma ima enak navidezni naslovni prostor kot starš. Kopija navideznega naslovnega prostora se izvede po mehanizmu COW. Dokler oba procesa (starš in otrok) le bereta iz tega naslovnega prostora, ni nobene potrebe po dejanskem kopiranju. Ob

prvem pisanju se potem naredijo dejanske kopije pomnilniških strani, ki se spreminjajo.

Mehanizem COW lahko uporabljajo podatkovne baze in datotečni sistemi pri implementaciji *trenutnih posnetkov* (angl. *snapshot*). Trenutni posnetek je konceptualno kopija celotnih podatkov v določeni točki v času, vrsta rezervne kopije (angl. *backup*). Dejansko kopiranje podatkov bi seveda vzelo ogromno časa in prostora, kar je nesprejemljivo. Namesto tega se naredijo samo reference na obstoječe podatke, pri čemer se reference označi kot reference COW. Ob morebitnih spreminjanjih delov podatkov v prihodnosti (po točki v času, ko je bil posnetek narejen), se naredi dejanska kopija dela podatkov, na kateri se tudi naredi sprememba. Prvotna kopija ostane nespremenjena – posledično tudi posnetek ostane nespremenjen.

Deduplikativni datotečni sistem *btrfs* omogoča COW (med drugim) na nivoju datotek. Ustvarimo lahko t.i. *lahko kopijo* datoteke, kjer se podatkovni bloki kopirajo le ob spremembi. Za izdelavo lahke kopije ukazu za kopiranje na sistemih Unix (`cp`) podamo opcijo `--reflink`.

Navkljub temu, da mehanizem COW pospeši izdelavo kopij na datotečnem sistemu, lahko zaradi same narave mehanizma uvede precejšno fragmentacijo datotek na trdem disku. Ob modifikaciji referenc COW se namreč ustvari nov blok, ki najverjetneje ne bo zapisan v bližini originalnega bloka. Kar pomeni, da bo nova kopija fragmentirana, posledično pa se lahko močno poveča dostopni čas. Problem je opisan v [5], kjer so predlagane tudi možne rešitve.

Poglavje 3

Obstoječi deduplikativni datotečni sistemi

V tem poglavju bomo pregledali nekaj obstoječih deduplikativnih datotečnih sistemov. Mnogi med njimi poleg deduplikacije omogočajo še kar nekaj ostalih naprednih možnosti, vendar se bomo mi omejili predvsem na deduplikativne lastnosti.

3.1 ZFS

ZFS omogoča trenutne posnetke in vmesno deduplikacijo na nivoju blokov. Bloki so spremenljive dolžine med 512 bajtov in 128 kilobajtov. Deduplikacijo moramo ročno vključiti (za vsak del datotečnega sistema posebej), ker je privzeto izključena. Enake bloke identificira s pomočjo vrednosti kriptografske zgoščevalne funkcije SHA-256, možno pa je vključiti tudi dodatno preverjanje bit po bitu.

Deduplikacijsko tabelo (preslikave zgoščenih vrednosti v informacije o blokih) zaradi hitrostnih razlogov hrani v pomnilniku (če tako nastavimo in dokler ga je dovolj na voljo). Pri veliki količini podatkov se torej zelo poveča potreba po pomnilniku. Groba ocena je, da potrebujemo približno 20 GB pomnilnika za 1 TB podatkov [10]. Možno je nastaviti, da ZFS uporablja

disk SSD (angl. *solid state drive*) kot drugonivojski predpomnilnik, kar lahko precej zmanjša potrebo po sistemskem pomnilniku.

Prvotno je bil ZFS zasnovan za OS Solaris pri podjetju Sun Microsystems (danes Oracle Corporation) in je namenjen operacijskim sistemom tipa Unix. Obstaja več implementacij ZFS. Zaradi nezdružljivosti njegove licence CDDL (angl. *Common Development and Distribution Licence*) z licenco GPL ni možna direktna vključitev v jedro Linux, zato je za OS Linux na voljo kot razširitev v obliki modula jedra (angl. *kernel module*), obstaja pa tudi implementacija ZFS v obliki datotečnega sistema FUSE (zfs-fuse).

ZFS omogoča tudi zaščito podatkov s kontrolno vsoto (angl. *checksum*), redundanco (v smislu podvajanja blokov) in možnost samodejnega popravila okvar podatkov zaradi strojnih napak, stiskanje in šifriranje podatkov, napredno predpomnjenje podatkov v sistemskem pomnilniku ali na diskih SSD ter še druge možnosti.

3.2 btrfs

Datotečni sistem *btrfs* (angl. *B-tree file system*) omogoča trenutne posnetke in naknadno kot tudi vmesno deduplikacijo na nivoju blokov ter mehanizem COW (glej poglavje 2.8). Bloki so fiksne dolžine, privzeto 4 KB. Naknadna deduplikacija se izvede s posebnimi orodji, ki tečejo v uporabniškem programskem prostoru in jih uporabnik požene, kadar želi. Tako naknadna, predvsem pa vmesna deduplikacija, sta še vedno močno v razvoju.

Tudi btrfs tako kot ZFS deduplikacijsko tabelo zaradi hitrostnih razlogov hrani v pomnilniku, zato se lahko pri uporabi vmesne deduplikacije potreba po sistemskem pomnilniku zelo poveča [15].

Btrfs omogoča mehanizem COW nad posameznimi datotekami. Če pri kopiranju datoteke uporabimo stikalo `--reflink`, bo btrfs ustvaril kopijo datoteke tipa COW. Brez uporabe omenjenega stikala se ustvari običajna kopija.

Razvoj btrfs se je začel v podjetju Oracle Corporation, namenjen je ope-

racijskim sistemom tipa Unix. Izdan je pod licenco GPL in že nekaj časa vključen v jedro Linux, vendar ob njegovi uporabi še vedno vsakič dobimo sporočilo, da je datotečni sistem "eksperimentalen", kar nakazuje, da je še vedno močno v razvoju. Za uporabo najnovejših možnosti potrebujemo tudi čim novejšo različico jedra Linux, ki morda tudi še ni označena kot stabilna. Diskovni format btrfs je sicer od avgusta 2014 označen kot stabilen in se verjetno ne bo več spreminjal [14].

Btrfs omogoča tudi stiskanje in šifriranje podatkov, zaščito podatkov s kontrolno vsoto (angl. *checksum*), redundanco in možnost samodejnega popravila morebitnih okvar podatkov zaradi strojnih napak, podenote (angl. *subvolumes*), prostorsko učinkovito shranjevanje majhnih datotek ter mnoge druge napredne možnosti.

3.3 lessfs

Lessfs je odprotokodni datotečni sistem tipa FUSE (glej poglavje 2.4). Omogoča vmesno deduplikacijo na nivoju blokov. Bloki so fiksne dolžine, privzeto 128 KB. Enake bloke identificira s kriptografsko zgoščevalno funkcijo *Tiger* (angl. *Tiger Hash*) z dolžino izvlečka 192 bitov. Zaledje za shranjevanje blokov in meta informacij je v obliki datotek na obstoječem datotečnem sistemu. Bloki se shranjujejo v eni ali več datotekah (odvisno od nastavitev zaledja), meta informacije pa v vgrajenih ključ-vrednost podatkovnih bazah kot npr. *Tokyo Cabinet* in *Berkeley Database*.

Preslikave prstnih odtisov blokov v informacije o blokih hrani sicer v datotekah (pravzaprav v ključ-vrednost podatkovnih bazah), a jih predpomni v pomnilniku, velikost predpomnilnika je nastavljiva.

Lessfs omogoča tudi transparentno stiskanje in šifriranje podatkov ter replikacijo. Koda je izdana pod licenco GPL.

3.4 opendedup

Opendedup je odprtokodni datotečni sistem v uporabniškem programskem prostoru razvit v programskem jeziku Java. Omogoča vmesno in naknadno deduplikacijo na nivoju blokov. Omogoča replikacijo, trenutne posnetke in obljublja skalabilnost. Bloki so lahko fiksne ali spremenljive dolžine. Čeprav razvit v Javi, je trenutno podprt le na 64 bitnih sistemih OS Linux, kjer uporablja ogrodje FUSE [19].

Poglavje 4

Implementacija deduplikativnega datotečnega sistema

4.1 Lastnosti

Odločiti smo se morali, ali deduplikativni datotečni sistem implementirati znotraj jedra ali v uporabniškem programskem prostoru. Naš datotečni sistem rešuje specifične naloge in tudi želimo si enostavne namestitve poleg obstoječe programske opreme. Kot smo že razložili v poglavju 2.3, je v takem primeru najboljša odločitev, da datotečni sistem implementiramo v uporabniškem programskem prostoru. Tudi praktično vsi obstoječi deduplikativni sistemi, ki specifično rešujejo le problem deduplikacije in morda še kak povezan problem, so implementirani v uporabniškem programskem prostoru. Ker bo namenjen operacijskim sistemom tipa Unix, bolj konkretno Linux, smo uporabili ogrodje FUSE, ki je de-facto standard za razvoj datotečnih sistemov v uporabniškem programskem prostoru v okoljih Linux.

Za razvoj smo izbrali programski jezik C, ker je to domoroden programski jezik za uporabo ogrodja FUSE kot tudi domoroden programski jezik za programiranje na OS Linux. Dodatno že samo dejstvo, da smo implementi-

rali datotečni sistem in ne neke višjenivojske aplikacije, govori v prid izbire nižjenivojskega in domorodnega jezika C.

Izbrali smo vmesni oz. sinhroni način deduplikacije. Pri našem primeru uporabe si inženirji želijo, da je prihranek prostora takojšni in ne želijo kasneje poganjati posebnega orodja za deduplikacijo. Pri vmesni deduplikaciji se hitrost dela z datotekami malo zmanjša, vendar inženirjev tipično ne moti, če razpakiranje imeniškega drevesa izvorne kode traja malenkost dlje časa. Prav tako je morebitno zmanjšanje hitrosti pri običajnem delu inženirja z izvornimi datotekami – urejanje in shranjevanje datotek – zanemarljivo oz. se ne opazi.

Odločili smo se za deduplikacijo na nivoju datotek. Datoteke z izvorno kodo so namreč večinoma majhne datoteke. Pri urejanju majhnih datotek se ne zgodi, da bi ob spremembi le majhnega dela datoteke nespremenjeni del datoteke bil zelo velik, pri čemer bi se splačalo delati deduplikacijo na nivoju blokov.

Datotečni sistem smo poimenovali *sndfs*, kar je kratica za angl. *Sndfs's Not Deduplicating Filesystem*. Kratico lahko poljubno mnogokrat rekurzivno razvijemo naprej. Imenovanje sledi zgledu kratice GNU, ki pomeni angl. *GNU's Not Unix*. V obeh primerih gre seveda (v večji meri) za šalo.

4.2 Arhitektura

4.2.1 Osnovni princip

Osnovni princip razvoja datotečnega sistema z ogrodjem FUSE je preprost. Implementirati moramo funkcije, ki ustrezajo siceršnjim operacijam nad datotečnimi sistemi oziroma ustreznim sistemskim klicem. Te funkcije se potem kličejo po principu povratnih klicev (angl. *callback*). Ob sistemskem klicu `write` se npr. izvede povratni klic funkcije, ki ima sledeči podpis:

```
int (*write)(const char *path, const char *buf, size_t len,
             off_t offset, struct fuse_file_info *fi);
```

V telesu te funkcije implementiramo akcije, ki naj se zgodijo ob sistemskem klicu `write`. Vidimo, da tukaj kot parameter ne dobimo datotečnega deskriptorja (angl. *file descriptor*), ampak kar pot (`path`) do datoteke, ki se piše. Tako kot pri sistemskem klicu `write` imamo tukaj kazalec na lokacijo podatkov (`buf`) in količino podatkov v bajtih (`len`). Dobimo pa še parameter `offset`, ki je odmik – koliko od začetka datoteke naj začnemo pisati. V parametru `fi` dobimo morebitne dodatne informacije o odprti datoteki. Ta povratni klic se pravzaprav ne preslika nujno samo iz sistemskega klica `write`, ampak lahko tudi posredno iz sistemskega klica `seek`, ki nastavi trenutni odmik pisanja v datoteki. Ni namreč ločenega povratnega klica za sistemski klic `seek`.

Kaj počnemo v povratnih klicih, je povsem naša prosta izbira. Lahko npr. podatke tudi pošljemo preko omrežja na nek način. Lahko bi npr. naredili datotečni sistem, ki bi ob pisanju podatkov v datoteko z imenom `send_192.168.0.42_11007)` naredil paket UDP s pripadajočimi podatki in paket poslal na naslov IP 192.168.0.42 in port 11007. Mi smo seveda razvili datotečni sistem, ki se obnaša, kot se datotečni sistemi običajno obnašajo, zato podatkov ne pošiljamo preko omrežja, ampak jih nekam shranjujemo.

Podatke lahko shranjujemo npr. kar na nek obstoječ datotečni sistem – na nek način, npr. v obliki datotek. Lahko bi jih sicer shranjevali tudi npr. na nek razdelek na trdem disku in sicer na surovi način, brez obstoječega datotečnega sistema. Večina datotečnih sistemov FUSE podatke shranjuje kar na obstoječi datotečni sistem, ker je tako enostavneje za uporabnike in morda tudi zato, ker je enostavneje za implementacijo. Tako smo se odločili tudi mi. Morebitna implementacija možnosti shranjevanja na surov razdelek na trdem disku je verjetno hitrostna optimizacija, ki bi jo lahko naredili kasneje.

4.2.2 Shranjevanje podatkov

Podatke, ki jih shranjujemo, delimo na dva dela: *meta podatke* in *dejanske podatke*. Meta podatki so npr. preslikave poti datotek v njihove številke

inode, attributi datotek (tip datoteke, uporabnik in skupina, velikost, čas spremembe itd), podatek o lokaciji blokov podatkov, ki pripadajo datoteki, itd.

4.2.3 Dejanski podatki

Dejanske podatke shranjujemo surovo v obliki datotek. Sndfs izvaja deduplikacijo na nivoju datotek, zato je en blok spremenljive dolžine in ga shranimo kar kot eno datoteko na obstoječem datotečnem sistemu. Iz imena datotek je razvidna številka bloka. Bloku številka n pripada datoteka z imenom `n` v imeniku, kjer shranjujemo bloke.

Prednost takega načina shranjevanja podatkovnih blokov je, da se brisanje datotek izvaja v času $O(1)$ in da se pri tem prostor tudi dejansko sprosti. Če bi namreč bloke pisali zaporedoma v neko datoteko, potem bi izbris naključnega bloka pomenil izbris nekje v sredini datoteke. Potem bi imeli dve možnosti. Lahko bi takoj sprostili prostor, ki ga je zasedal izbrisani blok tako, da vse bloke, ki sledijo, premaknemo za ustrezno število bajtov proti začetku datoteke. To bi pomenilo, da je brisanje lahko časovno zelo potratna operacija. Lahko pa bi enostavno samo prostor izbrisanega bloka označili kot sproščen in na voljo morebitnim novim blokom. Pri tem bi operacija brisanja sicer bila $O(1)$, vendar bi poraba prostora kljub brisanju samo rasla. Razen če vsake toliko v ozadju poženemo proces, ki sprosti prostor izbrisanih blokov, kar pa spet obremenjuje sistem.

4.2.4 Meta podatki

Meta podatki so strukturirani. Lahko bi sicer sami oblikovali format in strukturo zapisa v datoteke, a smo se raje odločili, da jih zapisujemo v podatkovno bazo *Berkeley Database* (v nadaljevanju *BD*). BD je vgrajena podatkovna baza. Ni relacijska – ne uporablja npr. poizvedbenega jezika SQL – temveč vse podatke shranjuje kot preslikave ključev v vrednosti. Pri shranjevanju za določen ključ specificiramo vrednost. Pri branju pa iščemo po ključu in kot

rezultat dobimo vrednost. Ključ in vrednost sta lahko poljubna niza bajtov. BD je kompaktna in tako prostorsko kot hitrostno učinkovita. Ne temelji na principu odjemalec strežnik, ampak je podatkovna baza v obliki deljene programske knjižnice, s katero se program poveže. Omogoča tudi napredne možnosti, kot so transakcije ACID[1] in replikacija.

Meta podatke shranjujemo v dve podatkovni bazi BD. Prva je `dirs.db`, ki vsebuje preslikave poti datotek v njihove številke inode. Ključ je torej absolutna pot do datoteke, vrednost pa številka inode. Druga je `inodes.db`, ki vsebuje preslikave številk inode v meta podatke o datoteki. Ključ je torej številka inode, vrednost pa struktura z meta podatki o datoteki. Imenik je le datoteka posebne vrste. Pomemben atribut je številka bloka. Prazne datoteke in imeniki nimajo pripadajoče številke bloka. Imeniki vsebujejo tudi informacijo o otrocih – vsebovanih datotekah. Meta podatki so torej predstavljeni s strukturo:

```
typedef struct {
    dev_t        dev;
    uint64_t     ino;     /* številka inode */
    mode_t       mode;    /* zaščita (branje, pisanje, izvajanje) */
    nlink_t      nlink;   /* število trdih povezav */
    uid_t        uid;     /* lastnik */
    gid_t        gid;     /* skupina */
    dev_t        rdev;
    uint64_t     size;    /* velikost v bajtih */

    struct timespec    atime;   /* čas zadnjega dostopa */
    struct timespec    mtime;   /* čas zadnje spremembe */
    struct timespec    ctime;   /* čas zadnje posodobitve inode */

    uint64_t blocknbr;

    uint64_t childs_len;   /* velikost polja 'childs' */
    uint64_t childs_num;
```

[1] angl. *Atomicity, Consistency, Isolation, Durability* – atomarnost, skladnost, izolacija, trajanje.

```
    uint64_t childs[];     /* številke inode otrok (če je to imenik) */
} InodeMeta;
```

Številka inode 1 je rezervirana za korenski imenik /. Številka inode 0 je rezervirana za meta podatke o samem datotečnem sistemu. V bazi `inodes.db` se številka inode 0 ne preslika v `struct InodeMeta`, temveč v `struct SndfsMeta`. Ta struktura vsebuje podatke o zadnji uporabljeni številki inode in o zadnji uporabljeni številki bloka.

```
typedef struct {
    uint64_t ino_max;
    uint64_t blocknbr_max;
} SndfsMeta;
```

Številke inode in številke blokov datotek, ki se zbrišejo, se (zaradi poenostavitve implementacije) ne uporabijo ponovno. Glede na to, da sta obe števili široki 64 bitov, jih verjetno zelo težko kdaj zmanjka.

V bazah BD lahko iščemo le po ključih (ali pa hkrati po obeh – ključu in vrednosti). Poleg tega je možno definirati t.i. sekundarne podatkovne baze, ki predstavljajo drugačen pogled na podatke v primarni podatkovni bazi, kar efektivno pomeni drugačen način iskanja.

Bazi `dirs.db` asociiramo sekundarno podatkovno bazo `s_dirs.db`, kjer namesto po celotnih poteh iščemo le po imenih datotek in dobimo kot rezultat kombinacije številk inode in pripadajoče poti, kjer pot vsebuje specificirano ime datoteke. To potrebujemo pri iskanju kandidatov za deduplikacijo, kot bomo videli v nadaljevanju.

Bazi `inodes.db` asociiramo sekundarno podatkovno bazo `s_inodes.db`, kjer iščemo po številki bloka in kot rezultat dobimo vse številke inode, ki referencirajo to številko bloka. To potrebujemo, da ugotovimo, ali je vsebina neke datoteke duplicirana in da v tem primeru po potrebi ob morebitnem pisanju oz. spremembi naredimo novo kopijo.

4.3 Principi delovanja

Logika datotečnega sistema je realizirana znotraj funkcij za povratne klice FUSE (glej poglavje 4.2.1) in funkcij, ki se posredno kličejo iz teh funkcij.

4.3.1 Ustvarjanje in branje datotek

Ob ustvarjanju novih datotek (`mknod`) in novih imenikov (`mkdir`) ustvarimo novo preslikavo poti datoteke v novo številko inode in preslikavo shranimo v bazo `dirs.db`. Prav tako ustvarimo novo preslikavo številke inode v meta podatke o datoteki oziroma imeniku in preslikavo shranimo v bazo `inodes.db`. V meta podatkih imenika, v katerem smo ustvarili novo datoteko ali imenik, posodobimo informacije o otrocih tega imenika.

Ob branju podatkov iz datoteke najprej v podatkovnih bazah ugotovimo številko bloka datoteke. Ker bloke shranjujemo na obstoječem datotečnem sistemu v nekem namenskem imeniku v obliki datotek, kjer je ime datoteke številka bloka, enostavno le še preberemo podatke iz te datoteke.

4.3.2 Pisanje datotek

Zapisovanje podatkov bomo opisali natančneje, ker se seveda procesi deduplikacije dogajajo pri pisanju. Na najvišjem nivoju zapisovanje poteka tako, kot to prikazuje sledeča psevdo koda:

```
Potek_zapisovanja():
  open();
  while(podatki) {
    write();
  }
  release();
```

Ob odprtju datoteke se kliče *open()*, potem se enkrat ali večkrat kliče *write()*, ob zaprtju datoteke se kliče *release()*. Kolikokrat se kliče *write()*, je odvisno od več dejavnikov. Lahko se kliče točno enkrat za vsak dejanski

sistemski klic `write`, pogosto pa večkrat. FUSE *write()* kliče z neko (nastavljivo) največjo količino podatkov; če je podatkov več, se izvede več klicev.

Spodnja psevdokoda prikazuje potek *write()*:

```
write(pot, podatki, odmik):
  if (Je to prvi write()?) {
      if (Ima datoteka ze blok?) {
          copy_on_write();
      }
      sicer {
          kandidati_za_dedup = dobi_kandidate();
      }
  }

  Glede na trenutne podatke, odstrani morebitne vnose iz
  kandidati_za_dedup. Zapomni si spremembo stevila kandidatov.

  if (Smo imeli kandidate, jih ni vec?) {
      Datoteki priredi nov blok, skopiraj podatke do tega odmika
      iz zadnjega odstranjenega kandidata.

      Pisi trenutne podatke od odmika naprej.
  }
  if (Smo imeli kandidate, se vedno jih imamo?) {
      Nicesar ne pisemo.
  }
  if (Ni bilo kandidatov?) {
      Pisi trenutne podatke od odmika naprej.
  }


copy_on_write():
  if (Obstojeci blok asociiran z vec kot eno stevilko inode?) {
      Alociraj nov blok za to datoteko.

      Skopiraj podatke iz starega v nov blok.
  }
```

```
release():
  if (Smo imeli kandidate in jih se vedno imamo?) {
      Vzemi podvojeni blok prvega kandidata.

      Asociiraj stevilko inode te datoteke s podvojenim blokom.
  }
```

Ob prvem *write()* najprej naredimo določene akcije pred pisanjem. Če datoteka še nima asociiranega bloka (ravnokar na novo ustvarjena datoteka), potem najprej poiščemo potencialne kandidate za deduplikacijo. Če datoteka že ima asociiran podatkovni blok, potem izvedemo morebiten mehanizem piši ob pisanju (*copy_on_write()*).

4.3.3 Kandidati za deduplikacijo

Kandidati za deduplikacijo so datoteke, ki imajo enako ime (pot je lahko seveda različna). Prav tako so kandidati za deduplikacijo datoteke, ki so trenutno odprte samo za branje.

Datoteke z enakim imenom so kandidati za deduplikacijo, ker je verjetnost, da imata dve datoteki z enakim imenom enako vsebino pri našem primeru uporabe (več kopij imeniških drevesnih struktur izvorne kode) zelo velika. Primer sta npr. datoteki: `switch/linux/drivers/net/macvlan.c` in `switch_newhw/linux/drivers/net/macvlan.c`. Recimo, da že imamo na datotečnem sistemu drevesno strukturo z jedrom Linux pod imenikom `switch`. Potem pa ustvarimo še en imenik `switch_newhw`, kamor odpakiramo iz nekega arhiva (npr. tar.xz). še eno kopijo istega jedra Linux. V tem procesu bo nastalo ogromno datotek, med drugim tudi `macvlan.c`. Obe datoteki imata enako ime, le da sta na drugačni imeniški poti. Kot rečeno, je verjetnost, da imata enako vsebino, zelo velika, zato tako datoteko naš datotečni sistem da na seznam kandidatov za deduplikacijo.

Datoteke, ki so trenutno odprte samo za branje, so kandidati za deduplikacijo pri pogostem primeru, ko naredimo kopijo obstoječe datoteke. Npr. z ukazom `cp linux-3.16.2.tar.xz backup_linux-3.16.2.tar.xz`. V tem

primeru se prva datoteka odpre v načinu samo za branje, druga datoteka se ustvari na novo in odpre v načinu samo za pisanje. Prva datoteka je torej kandidat za deduplikacijo. Če bi obstajal mehanizem, da program `cp` datotečnemu sistemu sporoči, da dejansko dela natančno kopijo prve datoteke, potem prva datoteka ne bi bila samo kandidat za deduplikacijo, ampak bi že takoj zagotovo vedeli, da lahko blok druge datoteke enostavno kaže (referenca) na blok prve datoteke.

Seznam kandidatov za deduplikacijo potem ob vsakem *write()* posodabljamo glede na podatke, ki prihajajo. Kandidat za deduplikacijo preneha biti kandidat, če se njegovi podatki ne ujemajo s podatki, ki naj bi jih zapisali. Ob vsakem *write()* naj bi pisali določeno količino podatkov na določenem odmiku v datoteki. Na točno tem odmiku in točno tako količino podatkov primerjamo z obstoječimi podatki vsakega kandidata. Če se podatki ujemajo, kandidat ostane, sicer ga zbrišemo s seznama. Podatke primerjamo bajt po bajtu.

Pri vsakem posodabljanju kandidatov za deduplikacijo si zapomnimo spremembo v številu kandidatov pred in po posodobitvi seznama kandidatov. Če smo pred posodobitvijo imeli kandidate in jih po posodobitvi še vedno imamo, potem v tem trenutku ne delamo ničesar, saj so še vedno obeti, da bo lahko novonastajajoča datoteka duplicirana z blokom podatkov neke obstoječe datoteke. Če smo imeli kandidate, pa smo ravnokar ob tem *write()* ugotovili, da jih sedaj ni več, potem vemo, da bo novonastajajoča datoteka imela svoj lasten nededupliciran blok. Ta blok ustvarimo in iz zadnjega odstranjenega kandidata skopiramo vse podatke, ki so do tega odmika bili enaki, v novo ustvarjen blok. Trenutni podatki se že razlikujejo, zato jih zapišemo v novo ustvarjen blok od odmika naprej. Če sploh ni bilo kandidatov, pa enostavno zapišemo trenutne podatke od odmika naprej.

Če ob vsakem *write()* ugotavljamo, da še vedno imamo kandidate, potem torej podatkov ne zapisujemo. V tem primeru bomo šele ob *release()* ugotovili, da smo stalno imeli vsaj enega kandidata in da torej lahko izvedemo deduplikacijo. V tem primeru enostavno asociiramo številko inode novonasta-

jajoče datoteke z obstoječo številko bloka prvega kandidata za deduplikacijo. Podatki datoteke so bili torej s tem deduplicirani.

4.3.4 Kopiranje ob pisanju

Ostane nam še primer, ko ob prvem *write()* ugotovimo, da datoteka že ima asociiran podatkovni blok. To se zgodi, kadar npr. nek proces odpre obstoječo datoteko za pisanje in jo spreminja: npr. piše na konec datoteke ali pa samo spreminja obstoječe podatke. V tem primeru pred dejanskim pisanjem v *copy_on_write()* preverimo, ali so obstoječi podatki te datoteke morda deduplicirani; z drugimi besedami, ali si ta datoteka deli številko bloka z neko drugo obstoječo datoteko. Če je to res, potem moramo pred dejanskim pisanjem najprej trenutni datoteki ustvariti nov podatkovni blok, ki je kopija prejšnjega, ki si ga je datoteka delila z drugimi. S tem smo torej izvedli mehanizem COW.

Poglavje 5

Primerjava z lessfs

V tem poglavju bom primerjali naš deduplikativni datotečni sistem *sndfs* z odprtokodnim deduplikativnim datotečnim sistemom *lessfs*. Za primerjavo z lessfs smo se odločili, ker si delita nekaj skupnih lastnosti. Po drugi strani pa se vseeno razlikujeta v nekaj ključnih lastnostih, ki so zanimive za primerjanje s stališča deduplikacije.

5.1 Konceptualna primerjava

Oba sta razvita v jeziku C za operacijski sistem Linux in oba sta datotečna sistema v uporabniškem programskem prostoru tipa FUSE. Tako sndfs kot lessfs izvajata vmesno oz. sinhrono deduplikacijo.

5.1.1 Nivo deduplikacije

Bistvena razlika je v nivoju, na katerem deduplikacijo izvajata. Lessfs je precej klasičen deduplikativen sistem; deduplikacijo izvaja na nivoju blokov. Ti so fiksne dolžine (privzeto 128 KB). Dolžina blokov je sicer nastavljiva, vendar kot je razbrati iz obstoječe dokumentacije, je privzeta dolžina najbolj primerna za večino potreb.

Sndfs deduplikacijo izvaja na nivoju datotek, kar sicer v splošnem spada v kategorijo spremenljive dolžine blokov. Kot smo omenili v razdelku 2.6,

se tak način deduplikacije imenuje tudi *Single Instance Storage*. Tak način deduplikacije je manj pogost kot deduplikacija na nivoju blokov in tudi zato nam je bil zanimiv za raziskavo in analizo. Pomemben razlog za izbiro deduplikacije na nivoju datotek je tudi namen našega datotečnega sistema oz. tipični in specifični primeri uporabe, za katere smo si ga zamislili – deduplikativni datotečni sistem za okolje, v katerem dela en ali več razvijalcev in se na podatkovnem mediju pogosto pojavlja več kopij iste imeniške drevesne strukture izvorne kode. Pri tem razvijalci spreminjajo le zelo majhno podmnožico datotek, ki so tudi relativno majhne velikosti. Pri teh primerih uporabe se nam je zdel tak pristop dovolj učinkovit. Deduplikacija na nivoju blokov se nam je za te namene zdela nepotrebna.

5.1.2 Način identifikacije enakih blokov

Lessfs bloke identificira oz. ločuje s kriptografsko zgoščevalno funkcijo Tiger z dolžino izvlečka 192 bitov. Večina deduplikativnih datotečnih sistemov, ki deduplikacijo izvajajo na nivoju blokov in uporabljajo kriptografske zgoščevalne funkcije za računanje prstnih odtisov blokov, uporablja SHA-1 ali SHA-256 ali kako drugo metodo zgoščanja iz iste družine. Zakaj so se pri lessfs odločili za zgoščanje Tiger, ni možno razbrati iz obstoječe dokumentacije. Domnevamo pa lahko, da morda zato, ker je bilo zgoščanje Tiger razvito za učinkovito računanje na 64 bitnih platformah[6], ki so danes vedno pogostejše. Ali pa morda zato, ker ima znanih manj kriptografskih napadov (pravzaprav nobenega resnega oz. učinkovitega za polno zgoščanje Tiger[7]); morda tudi zato, ker ga ni razvila NSA[1], kot to velja za SHA-1 in SHA-256. Primerjanja bajt po bajtu lessfs ne izvaja.

Ko novi bloki prihajajo, lessfs izračuna vrednost izvlečka Tiger za vsak blok in preveri, ali morda blok z enako vrednostjo izvlečka že obstaja. Blok shrani le, če ne najde bloka z enakim izvlečkom. Vrednosti izvlečkov predpomni v pomnilniku. Ker primerjanj bajt po bajtu ne izvaja, ampak samo

[1]National Security Agency – t.i. Agencija za nacionalno (ne)varnost Združenih držav Amerike

išče po shranjenih zgoščenih vrednostih Tiger, se s tem potencialno lahko izogne nepotrebnim vhodno-izhodnim operacijam, ki so lahko inherentno zamudnejše kot branje iz glavnega pomnilnika. V primerov, da ni zadetkov v predpomnilniku zgoščenih vrednosti, mora zgoščene vrednosti prebrati s podatkovnega medija (tipično trdi disk). V primeru, da ne najde enake zgoščene vrednosti, kot jo ima trenutni blok, mora seveda zgoščeno vrednost in blok shraniti na podatkovni medij.

Sndfs bloke (datoteke) ločuje s primerjanjem bajt po bajtu. Zgoščenih vrednosti oz. prstnih odtisov podatkov ne računa. Za tak pristop smo se odločili iz dveh razlogov. Prvi razlog je eksperimentalna radovednost – želeli smo videti, kako se tak pristop obnese. Želeli smo narediti drugače kot sndfs in potem oba pristopa primerjati. Drugi razlog je v tem, da se izkaže, da implementacija deduplikacije na nivoju datotek z ločevanjem bajt po bajtu ni nujno enostavnejša od implementacije ločevanja z zgoščenimi vrednostmi, kar še malo bolj velja, ko za razvoj uporabimo ogrodje FUSE.

Zgoščeno vrednost datoteke namreč lahko v celoti izračunamo šele, ko dobimo vse podatke te datoteke. Podatki prihajajo v splošnem lahko po kosih (glej razdelek 4.3.2) in pri velikih datotekah lahko nekaj časa traja, preden dobimo vse podatke. Dokler nimamo vseh podatkov datotek, torej tudi še nimamo zgoščene vrednosti datoteke, kar pomeni, da vse do konca ne vemo, ali so podatki duplicirani, kar pomeni, da jih moramo ta čas nekam shranjevati. Lahko bi jih shranjevali v pomnilnik, ker je to hitreje. Ampak v splošnem so lahko datoteke velike, kar lahko predstavlja nepotrebno ogromno porabo glavnega pomnilnika. Dodatno je problem, ker FUSE izvaja tudi več pisanj sočasno iz več niti, kar bi še dodatno povečalo porabo pomnilnika. Podatke bi torej ta čas morali shranjevati na podatkovni medij, kar pa zahteva vhodno-izhodne operacije, ki so lahko počasne. Če se odločimo, da namesto računanja zgoščene vrednosti raje primerjamo podatke bajt po bajtu, potem moramo tudi brati obstoječe podatke z diska in jih primerjati s podatki, ki prihajajo. Ampak v tem primeru nam prihajajočih podatkov ni treba shranjevati nikamor (ne v pomnilnik ne na podatkovni medij), vsaj dokler imamo

vsaj enega kandidata za deduplikacijo. V trenutku, ko morda ugotovimo, da nimamo več nobenega kandidata za deduplikacijo, moramo seveda podatke shraniti. Vmes jih res nismo nikamor (začasno) shranjevali, kar pa ni težava, saj smo do tega trenutka imeli vsaj enega kandidata za deduplikacijo. Zato v tem trenutku za novonastajajočo datoteko enostavno shranimo podatke tako, da jih skopiramo iz enega bloka ravnokar izgubljenega kandidata (glej 4.3.2). To so razlogi za primerjanje bajt po bajtu.

Lessfs ima torej *enonivojsko identifikacijo duplikatov* – odločitev na podlagi zgoščene vrednosti. Lessfs enostavno izračuna zgoščeno vrednost bloka (relativno majhne velikosti – privzeto 128 KB) in potem to vrednost primerja s shranjenimi vrednostmi. To iskanje potencialnih kandidatov se lahko učinkovito implementira z uporabo predpomnjenja in npr. uporabo podatkovne strukture *zgoščena tabela*, kar lessfs tudi počne. Po drugi strani sndfs podatke primerja bajt po bajtu. Seveda prihajajočih podatkov ne more kar primerjati bajt po bajtu z vsemi trenutnimi podatki na datotečnem sistemu, saj bi bilo to katostrofalno zamudno. Zato ima sndfs *dvonivojsko identifikacijo duplikatov*. Na prvem nivoju identificira potencialne kandidate na podlagi imen datotek in podatkov o trenutno odprtih datotekah na tem datotečnem sistemu. Kandidati za deduplikacijo so datoteke z enakim imenom in pa datoteke, ki so trenutno odprte samo za branje (glej razdelek 4.3.3).

5.1.3 Zaledje

Lessfs omogoča več vrst zaledij. Podatke lahko shranjuje v neko vgrajeno ključ-vrednost podatkovno bazo (Berkeley Database, Tokyo Cabinet ali Hamster DB) ali pa v datoteke. Odločili smo se, da vse primerjave z lessfs delamo z vključenim zaledjem Berkeley Database za meta podatke in zaledjem *file_io* za shranjevanje blokov. Ta nastavitev je namreč privzeta, ker je najbolj preizkušena in najbolje deluje za večino potreb. Pri zaledju file_io se bloki shranjujejo v eno samo datoteko.

Sndfs meta podatke shranjuje v vgrajeno ključ-vrednost podatkovno bazo Berkeley Database. Navdih za to odločitev smo dobili ravno pri lessfs. Po-

datkovne bloke sndfs shranjuje v obliki več datotek v namenskem imeniku. Ker se deduplikacija pri sndfs izvaja na nivoju datotek, en blok predstavlja eno datoteko, ki se shrani kot datoteka v namenskem imeniku, pri čemer je ime datoteke kar številka bloka.

Pri zaledju file_io se podatki shranjujejo v eno samo datoteko, zato je brisanje podatov pri lessfs v splošnem časovno potratna operacija. Pravo brisanje s sproščanjem porabljenega prostora namreč zahteva tudi premik vseh blokov v datoteki, ki sledijo izbrisanemu bloku. Ta problem lessfs rešuje tako, da ob brisanju datotek zbriše le reference na bloke, prostora pa dejansko ne sprosti. Količina zasedenega prostora torej stalno samo raste. Možno je vključiti, da se vsake toliko časa v ozadju požene proces, ki prostor dejansko sprosti, vendar je zaradi časovne zahtevnosti ta možnost privzeto izključena.

Sndfs po drugi strani datoteke briše tako časovno kot prostorsko učinkovito. Prostor se dejansko sprosti. To nam omogoča način shranjevanja podatkov v obliki datotek v namenskem imeniku.

5.1.4 Ostalo

Lessfs kot splošnonamenski deduplikativni datotečni sistem omogča še nekaj drugih možnosti, ki jih sndfs ne. Naj tukaj samo omenimo, da ima privzeto vključeno transparentno in sinhrono stiskanje podatkovnih blokov. To seveda lahko še dodatno pripomore k manjši porabi prostora. Je pa lahko časovno potratno. Pri razvoju sndfs smo se osredotočali samo na deduplikacijo kot proces inteligentnega podatkovnega stiskanja, zato dodatnega običajnega stiskanja nismo implementirali. Implementacija bi bila sicer relativno preprosta, saj bi samo uporabili kakšno obstoječo knjižnico z algoritmi za stiskanje. Vse eksperimentalne primerjave z lessfs smo seveda zato delali tako, da smo stiskanje podatkov izključili.

5.2 Eksperimentalna primerjava

Izvedli smo nekaj testov z lessfs in sndfs na realnih, pa tudi nerealnih podatkih in ju primerjali med seboj. Zanimala sta nas tako časovna kot prostorska učinkovitost (predvsem slednja).

Vse eksperimente smo delali ob izključenem avtomatskem stiskanju podatkov pri lessfs. Nastavljeno zaledje pri lessfs je bilo Berkeley Database za meta podatke in file_io za dejanske podatke. Ko smo kopirali podatke na testni datotečni sistem, je bil izvor podatkov vedno na drugem razdelku kot ponor oziroma zaledje testnega datotečnega sistema. Tako razdelek, s katerega smo kopirali, kot razdelek, na katerem je bilo zaledje testnega datotečnega sistema, sta bila formatirana z datotečnim sistemom ext4 s privzeto velikostjo bloka 4 KB. Ponorni datotečni sistem z zaledjem testnega datotečnega sistema je imel vedno okrog 90 odstotkov prostega prostora.

Eksperimente smo izvajali na operacijskem sistemu Linux različice jedra 3.11.0 na računalniku PC s procesorjem Intel Core 2 Duo E6550, 2.3 GHz, 4 MB predpomnilnika in 6 GB sistemskega pomnilnika RAM. Uporabljen je bil trdi disk WD 500 GB, 7200 rpm, Sata-II, 16 MB predpomnilnika.

Vplive predpomnjenja datotek v samem operacijskem sistemu na zaporedne meritve smo poskušali zmanjšati tako, da smo najprej nekaj časa eksperimentalne podatke čim večkrat zaporedoma brali in nekam zapisovali, dokler se nam je še zdelo, da se hitrost izboljšuje zaradi več in več podatkov v predpomnilniku. Meritve smo sicer večkrat ponovili in izračunali povprečje; vseeno so dobljene meritve časov precej približne ocene.

5.2.1 Imeniška drevesa izvorne kode

Opis eksperimenta

Na datotečni sistem smo zaporedoma posneli tri kopije imeniške drevesne strukture izvorne kode jedra Linux različice 3.16.2. Celotno drevo z izvorno kodo vsebuje 3046 imenikov in 47425 datotek (pred štetjem oz. eksperimentom smo odstranili manjše število posebnih datotek vtičnic). Večina izmed

njih so datoteke `.c` in `.h` z izvorno kodo v programskem jeziku C. Nekaj je tudi datotek Makefile in datotek z dokumentacijo. Vse naštete datoteke so tekstovne, t.j. vsebujejo tekst v človeku razumljivem formatu. Ostali tipi datotek so redki. Vsota velikost datotek (in imenikov) je 553172901 bajtov (ca. 528 MB). Zaradi organizacije podatkov po blokih podatki na datotečnem sistemu skoraj vedno zasedejo nekaj več prostora, kot pa je njihova velikost. Na datotečnem sistemu *ext4* tako celotna drevesna struktura zasede 664494080 bajtov (ca. 634 MB). Vsoto velikosti datotek smo merili z ukazom `du --block-size=1 -s --apparent-size linux-3.16.2`, oceno porabe prostora pa z ukazom `du --block-size=1 -s linux-3.16.2`. Povprečna velikost datoteke je (zaokroženo na bajt) 11397 bajtov, mediana pa 4139 bajtov.

Pred eksperimentom in po vsakem snemanju podatkov smo pomerili velikost meta podatkov in velikost dejanskih podatkov. Velikost smo merili s pomočjo programa `df` tako, da smo ga pognali pred in po snemanju podatkov. Program pokaže količino porabljenega in prostega prostora za vse razdelke vseh priklopljenih datotečnih sistemov na sistemu. Velikost meta in dejanskih podatkov smo merili tudi s programom `du`. Vedno smo merili porabo prostora na zalednem datotečnem sistemu (ext4) in ne vsote dejanske velikosti datotek.

Čas operacij smo merili s programom `time`. Ta izpiše čas, ki ga je proces preživel pri izvajanju v uporabniškem programskem prostoru, čas preživet med izvajanjem sistemskih klicev in skupni realni čas (ki vključuje tudi čakanje na vhodno-izhodne operacije). Seveda nas je zanimal slednji. Kot omenjeno v prejšnjem razdelku, so meritve časa precej približne ocene.

Rezultati in primerjava

V tabeli 5.1 so zbrani rezultati meritev za lessfs, v tabeli 5.2 pa rezultati meritev za sndfs. Za vsako meritev so zbrani podatki o velikosti meta podatkov in podatkov o blokih ter sprememba velikosti meta podatkov in podatkov o blokih. Navedeni sta tudi celokupna sprememba porabe prostora in čas

izvajanja operacije.

	začetek	meritev 1	meritev 2	meritev 3
meta	81.2 MB	142.5 MB	175.5 MB	198.0 MB
meta Δ		61.3 MB	33.0 MB	22.5 MB
bloki	0	445.6 MB	445.6 MB	445.6 MB
bloki Δ		445.6 MB	0 MB	0 MB
skupaj Δ		506.9 MB	33.0 MB	22.0 MB
čas		1m15s	0m53s	0m52s

Tabela 5.1: lessfs: Zaporedno snemanje več kopij imeniške drevesne strukture izvorne kode jedra Linux.

	začetek	meritev 1	meritev 2	meritev 3
meta	8.2 MB	59.9 MB	111.9 MB	194.1 MB
meta Δ		51.7 MB	52.0 MB	82.2 MB
bloki	0 MB	621.5 MB	621.5 MB	621.5 MB
bloki Δ		621.5 MB	0 MB	0 MB
skupaj Δ		673.2 MB	52.0 MB	82.2 MB
čas		1m46s	3m10s	6m45s

Tabela 5.2: sndfs: Zaporedno snemanje več kopij imeniške drevesne strukture izvorne kode jedra Linux.

Vidimo lahko, da lessfs že na začetku porabi približno 10 krat več prostora za meta podatke – 81 MB in 8 MB. Ko posnamemo imeniško drevesno strukturo jedra Linux, ki sicer na datotečnem sistemu ext4 zasede 634 MB bajtov, se meta podatki povečajo za 61,3 MB, bloki pa zasedejo le 445,6 MB. Skupna poraba prostora se torej poveča za 506,9 MB. Prihranek prostora na račun deduplikacije je torej 634 MB − 506.9 MB = 127,1 MB. Prihranek prostora še vedno ostane tudi, če od te številke odštejemo veliko začetno količino meta podatkov (81,2 MB). Ko pri drugem in tretjem koraku v dva ločena

imenika posnamemo spet enako imeniško drevesno strukturo, velikost podatkovnih blokov ostane enaka. Vsakič se povečajo le meta podatki. Zanimivo je, da vsakič za malo manj. Ena možna razlaga bi bila, da je to zaradi notranje strukture baze Berkeley Database, ki je nam neznana. Povečanje meta podatkov je seveda relativno precej majhno glede na celokupni prihranek prostora.

Lessfs torej že na prvem koraku prihrani prostor, kar je logično, saj dela deduplikacijo na nivoju blokov, pri tako veliki količini tovrstnih podatkov pa je pričakovano, da se najdejo kakšni podvojeni bloki že znotraj enega samega drevesa izvorne kode.

Pri sndfs je na začetku povečanje meta podatkov približno enako, potem pa očitno rahlo raste. Opazimo lahko sicer, da se do tretjega koraka količina meta podatkov pri sndfs in lessfs približno izenačita.

Podatkovni bloki pri sndfs zasedejo 621 MB. Tako kot pri lessfs tudi pri sndfs količina podatkovnih blokov ostane enaka tudi v drugem in tretjem koraku. Skupno povečanje prostora v prvem koraku je 673 MB, kar je več kot pa podatki sicer zasedejo na datotečnem sistemu ext4. V prvem koraku torej sndfs v tem primeru ne izkaže prihranka prostora. Kar je razumljivo glede na način, kako išče kandidate za deduplikacijo (enaka imena, trenutno odprte datoteke). V drevesu izvorne kode je 47425 datotek, od tega je 33387 unikatno poimenovanih. Največ datotek z enakim imenom predstavljajo datoteke `Makefile` ($n = 1871$), sledijo datoteke `Kconfig` ($n = 1083$), `.gitignore` ($n = 129$), `Kbuild` ($n = 114$), `irq.c` ($n = 102$), itd. Npr. datoteka z imenom `termios.h` se pojavi 30 krat in med njimi sta le dve med seboj enaki, ostale so si različne. Ti rezultati so torej pričakovani, saj smo sndfs zasnovali z namenom, da prihranek prostora izkaže pri vsaj dveh kopijah istih (ali podobnih) imeniških drevesnih struktur izvorne kode.

Pri sndfs podatkovne bloke shranjujemo v obliki datotek v namenskem imeniku, zato lahko potem, ko smo posneli točno eno kopijo imeniške drevesne strukture izvorne kode jedra Linux, enostavno preštejemo število datotek v tem namenskem imeniku in s tem dobimo število unikatnih datotek v izvorni

kodi jedra Linux. To število je 47127 (od skupno 47425 datotek).

Kot že na prvi pogled vidimo v tabeli, po drugem koraku sndfs dejansko prihrani prostor. Po tretjem koraku je skupno povečanje prostora 807.4 MB. Pri običajnem nededuplikativnem datotečnem sistemu ext4 bi pričakovali povečanje prostora za $3 \times 634 = 1902$ MB. Sndfs torej po treh korakih v tem primeru prihrani 1094.6 MB prostora.

Vidimo, da pri lessfs prvo snemanje podatkov traja 1 minuto in 15 sekund. Ostala dva koraka pa traja približno 52 s. Ker so pri drugem in tretjem koraku na sistemu že duplikati, jih lessfs ne shranjuje, ampak shranjuje le meta podatke. Na ta račun se verjetno zmanjša porabljen čas.

Čas snemanja podatkov pri sndfs pri prvem koraku traja 1 minuto in 46 sekund, kar je morda malce slabše kot pri lessfs, ampak še vedno zadovoljivo, sploh če upoštevamo, da nismo delali nobenih optimizacij. Presenetili pa so nas časi v drugem in tretjem koraku, ki enormno narastejo.

Domnevamo, da razlog ni morda toliko v količini branj podatkov z diska, ko se primerjajo podatki morebitnih kandidatov za deduplikacijo, ampak v precej naključnem dostopu do podatkov (zaradi načina, kako sndfs shranjuje podatke), kar povzroči, da enkrat z diska preberemo manjšo količino tukaj, potem nekje drugje, potem spet na istem (ali bližnjem) mestu kot prej. Kot vemo, je glavna zakasnitev pri diskih v času dostopa. Povprečni čas dostopa pri uporabljenem disku je 8,9 ms, maksimalna hitrost branja podatkov pa 300 MB/s. V 8,9 ms časa, ki gre v nič, bi lahko prebrali že 2,67 MB podatkov. Teorija je še toliko bolj verjetna, ker vemo, da imamo veliko majhnih datotek in torej večino časa beremo precej majhne količine podatkov naenkrat. Če ta teorija drži, bi se ta problem v veliki meri verjetno dalo rešiti s predpomnjenjem branj blokov kandidatov, morda pa tudi s predpomnjenjem pisanj samih podatkovnih blokov.

5.2.2 Imeniška drevesa izvorne kode – naključni podatki

Pri prvem eksperimentu smo ugotovili, da sndfs pri prvem koraku še ni izkazal prihranka prostora, lessfs pa že. Kot smo že omenili, je razlaga za to v tem, da lessfs izvaja deduplikacijo po blokih in pri tako velikih količini podatkov zagotovo najde nekaj enakih blokov. Da bi to popolnoma potrdili, smo se odločili narediti eksperiment, kjer smo uporabili naključne podatke, ki jih praktično ni mogoče deduplicirati. Da bi bil eksperiment analogen prvem eksperimentu, smo si pripravili popolnoma enako imeniško drevesno strukturo izvorne kode jedra Linux, le da smo vsebino vseh datotek nadomestili z naključnimi podatki.

Naključne podatke smo pridobili iz naprave `/dev/urandom` na OS Linux. `/dev/urandom` je posebna naprava, ki ob branju daje psevdonaključne podatke. Podobna naprava je `/dev/random`, ki tudi daje psevdonaključne podatke, le da ta lahko blokira, če v nekem trenutku ni na voljo dovolj entropije iz okolja. `/dev/urandom` nikoli ne blokira, ker lahko že uporabljen šum iz okolja ponovno uporabi za entropijo. `/dev/urandom` smo uporabili, ker potrebujemo veliko naključnih podatkov v relativno kratkem času in ker se za naše potrebe popolnoma zadovoljimo s stopnjo naključnosti, ki jo ponuja.

Test s sndfs je pričakovano dal povsem enake rezultate kot pri normalnih, nenaključnih podatkih v prvem eksperimentu. Rezultati meritev so praktično enaki tistim v tabeli 5.2.

Test z lessfs pa je dal drugačne rezultate kot test pri prvem eksperimentu, ampak tudi ti rezulati so bili pričakovani. Če pogledamo v tabelo 5.3, vidimo, da meta podatki rastejo enako kot pri prvem eksperimentu. Porast podatkov za bloke pri prvem snemanju pa je 527 MB, kar je več kot pri prvem eksperimentu. Vrednost 527 MB pa je tudi praktično enaka vsoti velikosti vseh datotek (in imenikov) imeniške drevesne strukture izvorne kode Linux. To smo tudi pričakovali, saj imamo naključne podatke, ki se težko deduplicirajo.

Vidimo tudi, da so časovni parametri približno enaki, kot pri prvem eksperimentu.

	začetek	meritev 1	meritev 2	meritev 3
meta	81.2 MB	142.4 MB	175.8 MB	198.2 MB
meta Δ		61.2 MB	33.4 MB	22.4 MB
bloki	0	527.5 MB	527.5 MB	527.5 MB
bloki Δ		527.5 MB	0 MB	0 MB
skupaj Δ		588.7 MB	33.4 MB	22.4 MB
čas		1m12s	0m52s	0m48s

Tabela 5.3: lessfs: Zaporedno snemanje več kopij imeniške drevesne strukture izvorne kode jedra Linux, kjer je bila vsebina vseh datotek zamenjana z naključnimi podatki.

5.2.3 Video datoteke

Tretji eksperiment smo naredili tako, da smo na testni datotečni sistem zaporedoma večkrat posneli video datoteko v formatu *divx*. Datoteko smo snemali vsakič v nov imenik in enako poimenovano – tako smo bili prijazni do datotečnega sistema sndfs, ki kandidate za deduplikacijo izbira na podlagi imen datotek. Izbrana video datoteka v formatu divx je velikosti 994473984 bajtov (948,4 MB), na datotečnem sistemu ext4 pa zasede le 6 KB več – 994480128 bajtov (948.4 MB). Podatki v formatu divx so seveda stisnjeni podatki, zato tukaj deduplikacije ne pričakujemo.

	začetek	meritev 1	meritev 2	meritev 3
meta	81.2 MB	84.7 MB	86.5 MB	87.7 MB
meta Δ		3.5 MB	1.8 MB	1.2 MB
bloki	0	952.1 MB	952.1 MB	952.1 MB
bloki Δ		952.1 MB	0 MB	0 MB
skupaj Δ		955.6 MB	1.8 MB	1.2 MB
čas		0m33s	0m9s	0m11s

Tabela 5.4: lessfs: Zaporedno snemanje več kopij filma v formatu divx.

	začetek	meritev 1	meritev 2	meritev 3
meta	8.2 MB	8.2 MB	8.2 MB	8.2 MB
meta Δ		0 MB	0 MB	0 MB
bloki	0 MB	948.4 MB	948.4 MB	948.4 MB
bloki Δ		948.4 MB	0 MB	0 MB
skupaj Δ		948.4 MB	0 MB	0 MB
čas		0m12s	0m14s	0m5s

Tabela 5.5: sndfs: Zaporedno snemanje več kopij filma v formatu divx.

Kot vidimo v tabeli 5.4 meta podatki pri datotečnem sistemu lessfs sicer rastejo, a precej počasneje kot pri prvih dveh eksperimentih. Kar je pričakovano – pri eni datoteki je meta podatkov verjetno res manj kot pri ogromno datotekah.

Vidimo tudi, da se pri prvem snemanju količina podatkovnih blokov poveča za 952,1 MB, kar je 3,7 MB več, kot je velikost datoteke. Tega povečanja za 3,7 MB si sicer ne znamo povsem razložiti. Morda je posledica načina, kako lessfs notranje organizira datoteko, v katero shranjuje podatkovne bloke. Količina podatkovnih blokov se pri nadaljnih snemanjih ne povečuje (nit za bajt).

Rezultati za sndfs so zbrani v tabeli 5.5. Vidimo, da se količina meta podatkov praktično ne spreminja (kar je bolje kot pri lessfs). Pravzaprav se je pri prvem snemanju količina meta podatkov povečala za točno 12 KB, kar smo v tabeli zaokrožili na 0 MB. Pri vseh ostalih snemanjih je količina meta podatkov ostala do bajta enaka. To seveda lahko razložimo z dejstvom, da za shranjevanje meta podatkov uporabljamo bazo Berkeley Database, ki očitno že na začetku alocira nekaj prostora v naprej (8,2 MB), kar je več kot dovolj za meta podatke kar nekaj datotek.

Količina podatkovnih blokov se pri sndfs pri prvem snemanju poveča za 994484224 bajtov (948.4 MB), kar je točno 4 KB več, kot sicer datoteka zaseda na datotečnem sistemu ext4. Tudi tukaj se sndfs izkaže bolje kot

lessfs. Tudi pri vseh nadaljnih snemanjih količina podatkovnih blokov ostaja ves čas enaka (do bajta natančno).

Kar se tiče časovnih parametrov, sta se oba datotečna sistema odrezala približno enako, morda bi lahko rekli, da sndfs celo malenkost bolje.

5.2.4 Brisanje datotek

Preizkusili smo tudi, kako se lessfs in sndfs obneseta pri brisanju datotek.

Brisanje video datoteke iz prejšnjega eksperimenta oba izvajata približno enako hitro (skoraj instantno). Lessfs prostora sicer ne sprosti v zaledju. Sndfs prostor sprosti.

Brisanje imeniških drevesnih struktur izvorne kode je malo daljše opravilo, zato smo proces brisanja testirali. Najprej smo posneli eno drevo izvorne kode, nato še drugo. To je bil začetek meritve, kjer smo pomerili količino meta podatkov in količino podatkov za bloke. Potem smo zbrisali eno kopijo, ter merili čas in količino meta podatkov in podatkov za bloke. Zatem smo zbrisali še drugo kopijo in merili enake parametre. Rezultati so zbrani v tabelah 5.6 in 5.7.

	začetek	brisanje 1	brisanje 2
meta	177.6 MB	175.0 MB	177.2 MB
bloki	527.1 MB	527.1 MB	527.1 MB
čas		0m30s	0m32s

Tabela 5.6: lessfs: Brisanje dveh imeniških drevesnih struktur jedra Linux.

	začetek	brisanje 1	brisanje 2
meta	110.8 MB	180.5 MB	144.4 MB
bloki	621.5 MB	621.5 MB	0 MB
čas		0m14s	0m15s

Tabela 5.7: sndfs: Brisanje dveh imeniških drevesnih struktur jedra Linux.

Iz rezultatov se lepo vidi, da lessfs podatkovnih blokov nikoli ne zbriše. Po drugem brisanju namreč na datotečnem sistemu nimamo več nobene kopije podatkov, zato bi pričakovali porabo prostora za bloke 0, pa temu ni tako. Sndfs se po drugi strani obnaša pričakovano in bloke ob drugem brisanju dejansko sprosti.

Vidimo tudi, da je pri brisanju naš datotečni sistem sndfs tudi približno dva krat hitrejši kot lessfs.

Poglavje 6

Sklepne misli in ugotovitve

6.1 Rezultati eksperimentov

Glede na to, kakšne prednosti naj bi imel tako splošen način deduplikacije, kot je deduplikacija na nivoju blokov in identifikacija s kriptografskimi zgoščevalnimi funkcijami, ki jo izvaja lessfs, lahko ugotavljamo, da se naš deduplikativni datotečni sistem sndfs, ki deluje na nivoju datotek in kandidate za deduplikacijo identificira na podlagi imen datotek in informacije o trenutno odprtih datotekah s stališča porabe oziroma prihranka podatkovnega prostora obnaša precej dobro oziroma primerljivo z lessfs. Kakor se lahko morda komu zdi, da način identifikacije kandidatov na podlagi imen datotek ne bi prinesel velike deduplikacije, pa se v našem konkretnem primeru uporabe (veliko enakih ali podobnih imeniških drevesnih struktur izvorne kode) zelo dobro oziroma dovolj dobro obnese. Torej tako, kot smo predvidevali pri zasnovi rešitve.

Zanimivo je, da smo odkrili, da se sndfs obnese bolje kot lessfs, kadar imamo opraviti z velikimi datotekami (npr. video datoteke) in ne kadar imamo opravka z veliko majhnimi datotekami, kar je bil osnovni primer uporabe, iz katerega smo izhajali pri zasnovi arhitekture lessfs. To je seveda enostavno razložljivo. Pri deduplikaciji na nivoju blokov imamo za vsak blok še nekaj metapodatkov (npr. njegovo zgoščeno vrednost), pri deduplikaciji

na nivoju datotek (kot jo to dela sndfs) imamo meta podatke le za datoteko.

Malo nas je presenetilo, da se sndfs s stališča hitrosti obnese slabše pri snemanju imeniških dreves izvorne kode. Za ta pojav smo predlagali nekaj možnih razlag in nekaj možnih rešitev.

Po drugi strani pa smo ugotovili, da se sndfs obnese zelo dobro in precej bolje kot lessfs pri brisanju datotek. Bolje se obnese tako s prostorskega kot časovnega stališča. Pri sndfs se podatki dejansko zbrišejo, pri lessfs pa privzeto kar ostajajo. Sndfs briše datoteke približno dvakrat hitreje kot lessfs. Ti rezultati so bili pričakovani, saj je bil to eden izmed ciljev pri snovanju arhitekture sndfs.

6.2 Možnosti izboljšav

Postavili smo teorijo, da je vzrok problema počasnega pisanja oz. počasne deduplikacije pri pisanju druge in vseh nadaljnih kopij imeniških drevesnih struktur izvorne kode najverjetneje zaradi dolgih iskalnih časov trdega diska, ti pa se pogosto pojavljajo zaradi same zasnove datotečnega sistema. Predlagali smo, da bi bilo verjetno smotrno poskusiti problem rešiti oz. optimizirati s predpomnjenjem branj blokov kandidatov in morda tudi še s predpomnjenjem pisanj samih podatkovnih blokov.

Možnost izboljšave bi bila še, da bi bilo zaledje shranjevanja podatkovnih blokov iz dveh delov: En del bi bil za relativno velike datoteke. Te bi se shranjevale tako, kot to trenutno počne sndfs, torej v obliki datotek v nekem namenskem imeniku. Relativno majhne datoteke pa bi se shranjevale v eno samo namensko datoteko. V primerih, ko imamo zelo veliko količino zelo majhnih datotek (kot je to npr. pri izvorni kodi jedra Linux) bi na ta način zelo zmanjšali nepotrebno porabo prostora zaradi blokovno orientiranega shranjevanja datotek na zalednem datotečnem sistemu. Datoteka namreč vedno dejansko zasede večkratnik velikosti bloka prostora. Pri tem bi za velike datoteke ohranili hitro brisanje, za majhne datoteke pa bi morda lahko odkrili kakšne pametne hevristike dejanskega sproščanja prostora.

Način, ko sndfs išče potencialne kandidate za deduplikacijo iz podatkov o trenutno odprtih datotekah samo za branje, deluje le, kadar so ti kandidati na istem datotečnem sistemu. Če na deduplikativnem datotečnem sistemu že imamo neko datoteko z imenom A in to datoteko skopiramo v datoteko z imenom B, potem sistem z iskanjem kandidatov preko informacij o trenutno odprtih datotekah dobro deluje. Ne deluje pa, če bi imeli datoteko A še na drugem datotečnem sistemu ali razdelku in to datoteko posneli na naš deduplikativni datotečni sistem pod nekim drugim imenom, recimo C. Tu oba pristopa za iskanje kandidatov odpovesta.

Možna rešitev bi bila uvedba še enega načina iskanja kandidatov in sicer iz prstnega odtisa (zgoščene vrednosti) prvih nekaj KB datoteke. Rešitev izhaja iz predpostavke, da sta datoteki, ki imata npr. prvih 8 KB enakih, zelo verjetno enako vsebino. Datoteke imajo sicer pogosto vključena zaglavja, ki so povsod enaka, vendar nihče pri zdravi pameti z dolžino teh zaglavij ne pretirava. Potrebno bi bilo le ugotoviti tisto pravo velikost podatkov na začetku datoteke, ki naj se uporabi za izračun zgoščene vrednosti. Ta zgoščena vrednost bi se potem shranila poleg bloka datoteke. Ko bi na datotečni sistem pisali datoteko A iz drugega razdelka, ampak pod imenom C, bi izračunali prstni odtis začetnih nekaj KB in ta prstni odtis potem poiskali v bazi že shranjenih prstnih odtisov. Če tak prstni odtis najdemo, potem smo dobili novega kandidata. Naprej nadaljne podatke primerjamo po vrednosti bajt po bajtu. Če takega prstnega odtisa ne najdemo, pomeni, da kandidata nimamo.

Morebiti bi bilo dobro razmisliti tudi o odpravi zaledja Berkeley Database in zasnovi lastnega zaledja za meta podatke. Tako bi imeli več kontrole nad porabo metapodatkov. Možno bi bilo tudi za zaledje uporabiti surov razdelek. Tako bi se izognili enemu nivoju, ko imamo spodaj še vedno nek gostiteljski datotečni sistem. To bi lahko pomenilo tako performančne izboljšave kot izboljšave lastnosti datotečnega sistema. Verjetno bi se tako dalo lažje realizirati hitro brisanje z dejanskim sproščanjem prostora.

6.3 Pogled v prihodnost

Vsekakor bo zanimivo opazovati, kako se bo deduplikacija razvijala v prihodnjih nekaj letih in kakšni novi deduplikativni sistemi se bodo pojavljali v teoriji in v uporabi. Ali se bo kljub vedno cenejšim in vedno zmogljivejšim podatkovnim nosilcem deduplikacija še naprej uporabljala pri izdelavi varnostnih kopij? Kaj se bo zgodilo z deduplikativnimi datotečnimi sistemi v uporabniškem programskem prostoru? Se bodo še uporabljali, ko oz. če bodo "pravi" jedrni deduplikativni datotečni sistemi postali stabilnejši, bolj testirani in bolj razširjeni? Tu mislimo seveda predvsem na ZFS in btrfs. Ali bodo deduplikacijo uporabljali tudi običajni uporabniki? Menimo, da ni daleč čas, ko bo vsak povprečen uporabnik računalnika poznal dva načina kopiranja datotek: ustvarjanje običajnih kopij in ustvarjenje kopij tipa kopiraj ob pisanju.

Literatura

[1] K. Jin, E. L. Miller: "The effectiveness of deduplication on virtual machine disk images". V *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ACM New York, ZDA, 2009.

[2] T. Yang, J. Zhang, W. Sun: "Alternatives for Eliminating Duplicate in Data Storage". V *International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013)*, Atlantis Press, str. 565–568, 2013.

[3] C. Policroniades, I. Pratt: "Alternatives for Detecting Redundancy in Storage Systems Data". V *USENIX Annual Technical Conference, General Track*, str. 73–86, 2004.

[4] M. O. Rabin: "Fingerprinting by Random Polynomials". V *Technical Report TR-15-81*, Center for Research in Computing Technology, Harvard University, 1981.

[5] Zachary Nathaniel Joseph Peterson: "Data placement for copy-on-write using virtual contiguity". Doktorska disertacija, University of California, Santa Cruz, 2002.

[6] R. Anderson, E. Biham: "Tiger: A fast new hash function.". V *Fast Software Encryption*, Springer Berlin Heidelberg, str. 89–97, 1996.

[7] F. Mendel, V Rijmen: "Cryptanalysis of the Tiger hash function". V *In Advances in Cryptology–ASIACRYPT*, Springer Berlin Heidelberg, str. 536–550, 2007.

[8] eXdupe.com: “Risk of hash collisions in data deduplication”. http://www.exdupe.com/collision.pdf, december 2010, (dostopano 17.9.2014).

[9] Jeff Bonwick: “ZFS Deduplication (Jeff Bonwick’s Blog)”. https://blogs.oracle.com/bonwick/entry/zfs_dedup, november 2009, (dostopano 17.9.2014).

[10] Constantin Gonzalez: “ZFS Deduplication: To Dedupe or not to Dedupe...”. http://constantin.glez.de/blog/2011/07/zfs-dedupe-or-not-dedupe, julij 2011, (dostopano 17.9.2014).

[11] Wikipedia: “Data deduplication”. http://en.wikipedia.org/wiki/Data_deduplication, (dostopano 17.9.2014).

[12] Wikipedia: “Rabin fingerprint”. http://en.wikipedia.org/wiki/Rabin_fingerprint, (dostopano 17.9.2014).

[13] Wikipedia: “Wear leveling”. http://en.wikipedia.org/wiki/Wear_levelling, (dostopano 17.9.2014).

[14] Wikipedia: “Btrfs”. http://en.wikipedia.org/wiki/Btrfs, (dostopano 17.9.2014).

[15] btrfs Wiki: “Deduplication”. https://btrfs.wiki.kernel.org/index.php/Deduplication, (dostopano 17.9.2014).

[16] IBM: “Develop your own filesystem with FUSE”. http://www.ibm.com/developerworks/linux/library/l-fuse/, (dostopano 17.9.2014).

[17] Sourceforge: “Filesystem in Userspace”.
http://sourceforge.net/p/fuse/wiki/OperatingSystems/,
(dostopano 17.9.2014).

[18] Sourceforge: “fuse: FUSE API Documentation”.
http://fuse.sourceforge.net/doxygen/index.html,
(dostopano 17.9.2014).

[19] “Opendedup”.
http://opendedup.org/, (dostopano 17.9.2014).

[20] “Dokan”.
http://dokan-dev.net/en/about/, (dostopano 17.9.2014).